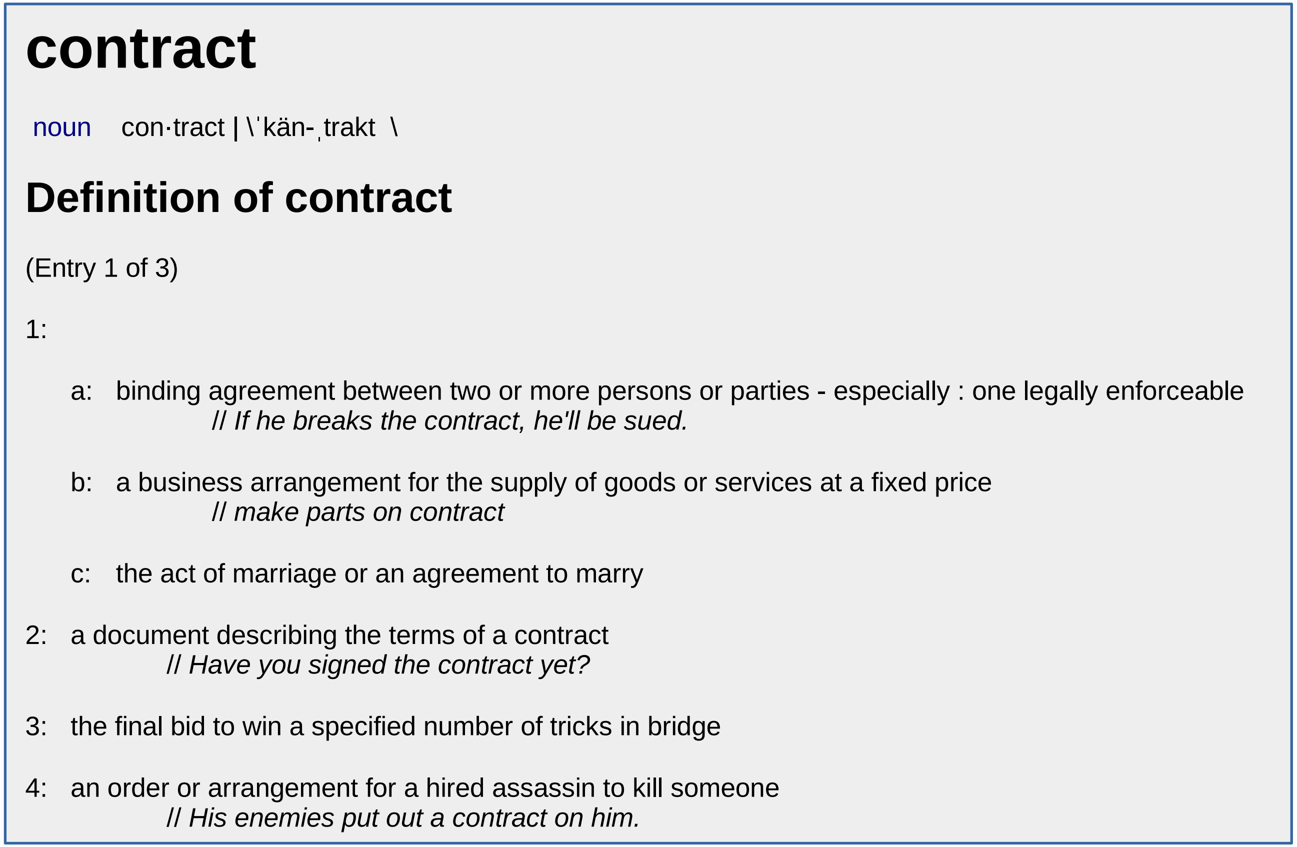
Programming with Contracts in C++20

🤝

Björn Fahller

What is a contract?



contract

noun con·tract | \ˈkän-ˌtrakt \

Definition of contract

(Entry 1 of 3)

1:

a: binding agreement between two or more persons or parties - especially : one legally enforceable
// If he breaks the contract, he'll be sued.

b: a business arrangement for the supply of goods or services at a fixed price
// make parts on contract

c: the act of marriage or an agreement to marry

2: a document describing the terms of a contract
// Have you signed the contract yet?

3: the final bid to win a specified number of tricks in bridge

4: an order or arrangement for a hired assassin to kill someone
// His enemies put out a contract on him.

`https://www.merriam-webster.com/dictionary/contract`

What is a contract?

contract

noun   con·tract | \ˈkän-ˌtrakt \

Definition of contract

(Entry 1 of 3)

1:

a: binding agreement between two or more persons or parties - especially : one legally enforceable
// If he breaks the contract, he'll be sued.

b: a business arrangement for the supply of goods or services at a fixed price
// make parts on contract

c: the act of marriage or an agreement to marry

2: a document describing the terms of a contract
// Have you signed the contract yet?

3: the final bid to win a specified number of tricks in bridge

4: an order or arrangement for a hired assassin to kill someone
// His enemies put out a contract on him.

`https://www.merriam-webster.com/dictionary/contract`

In SW design:

A formalized agreement, regarding program correctness, between a user and the implementation of a component.

What is a contract?

contract

noun con·tract | \ˈkän-ˌtrakt \

Definition of contract

(Entry 1 of 3)

1:

a: binding agreement between two or more persons or parties - especially : one legally enforceable
// If he breaks the contract, he'll be sued.

b: a business arrangement for the supply of goods or services at a fixed price
// make parts on contract

c: the act of marriage or an agreement to marry

2: a document describing the terms of a contract
// Have you signed the contract yet?

3: the final bid to win a specified number of tricks in bridge

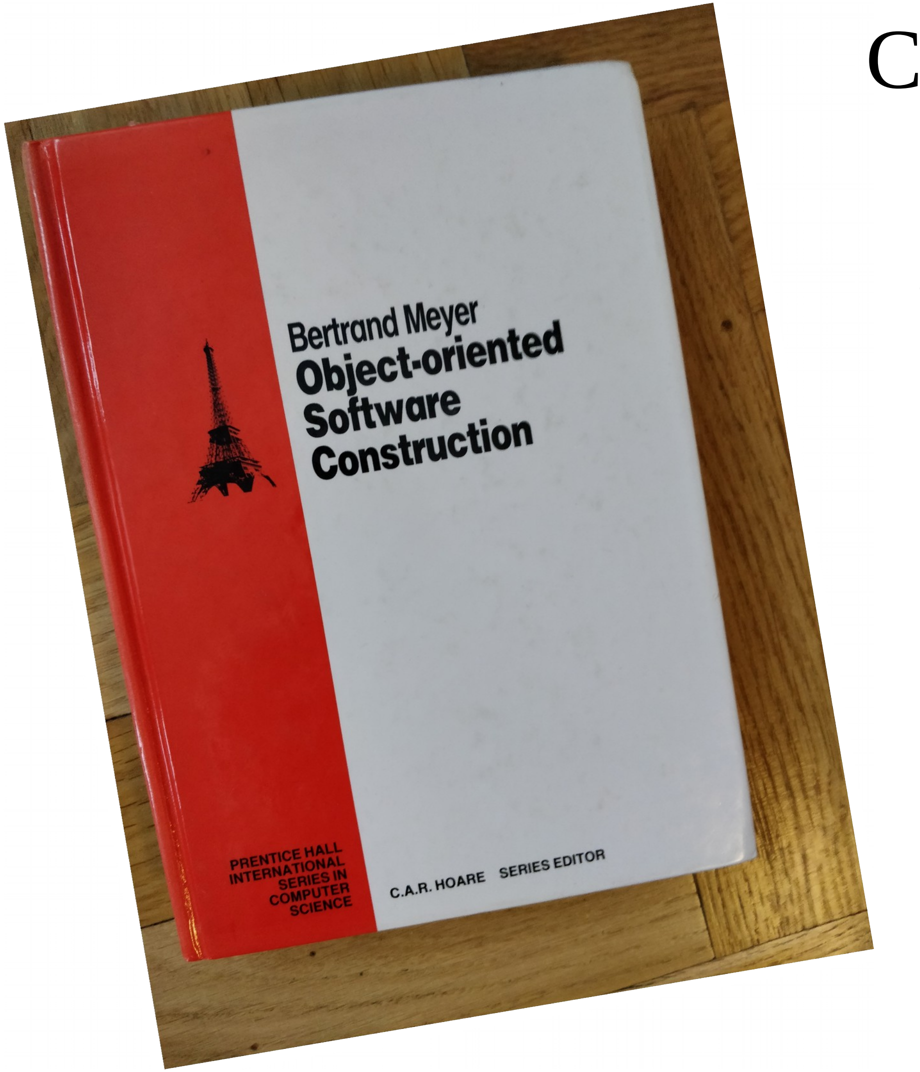
4: an order or arrangement for a hired assassin to kill someone
// His enemies put out a contract on him.

In SW design:

A formalized agreement, **regarding program correctness**, between a user and the implementation of a component.

https://www.merriam-webster.com/dictionary/contract

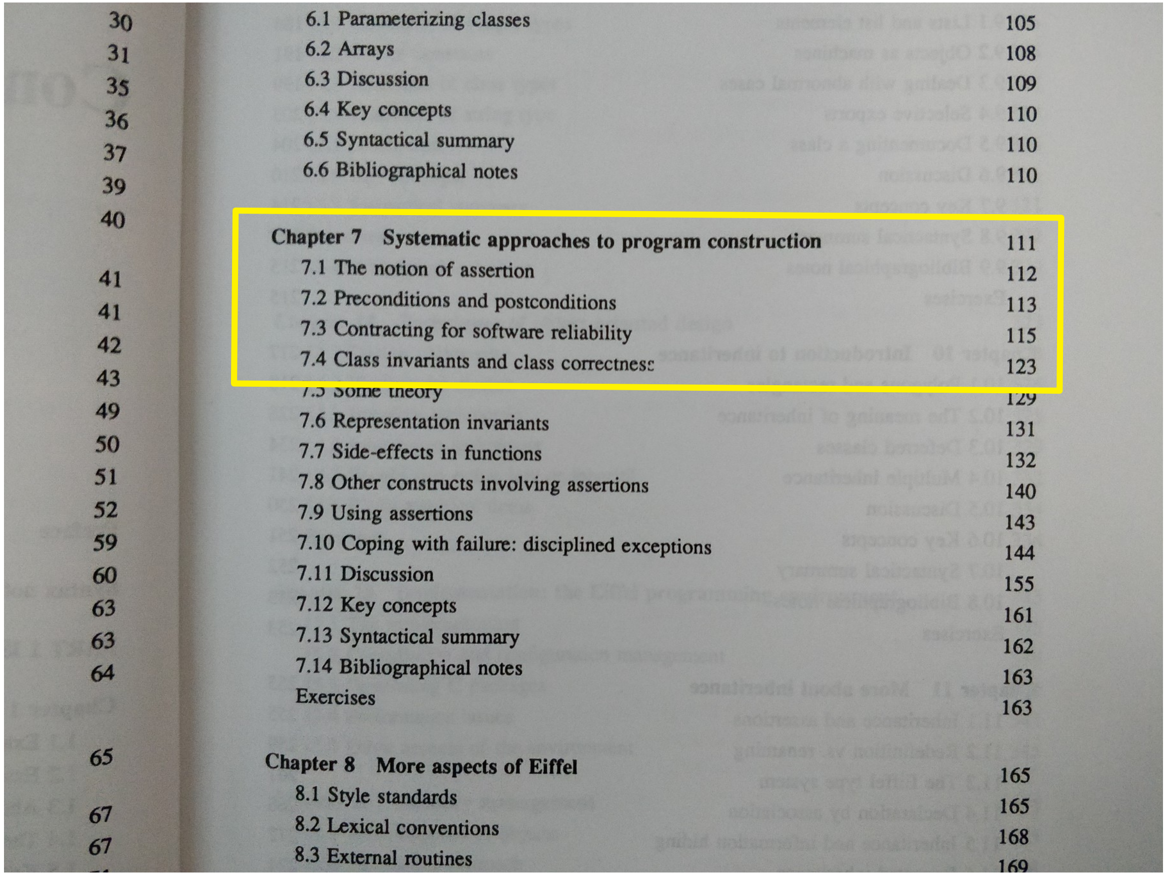
Contracts


Bertrand Meyer
Object-oriented Software Construction
PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR

- Object-oriented Software Construction
  - Bertrand Meyer - 1988
  - ISBN 978-0136290490

Contracts

- Preconditions
- Postconditions
- Class invariants

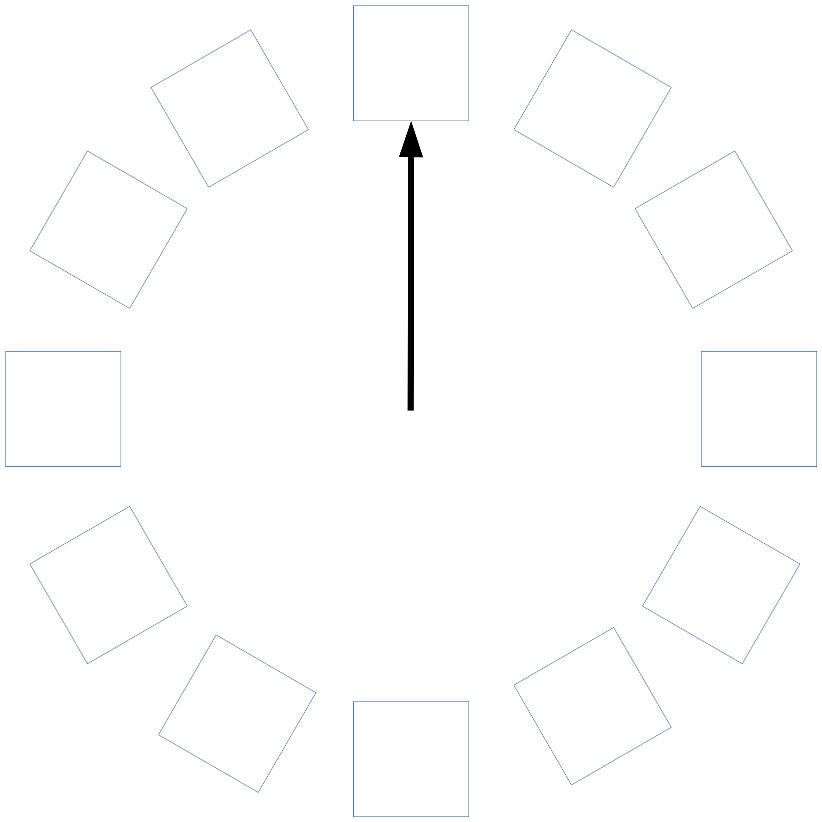


Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```

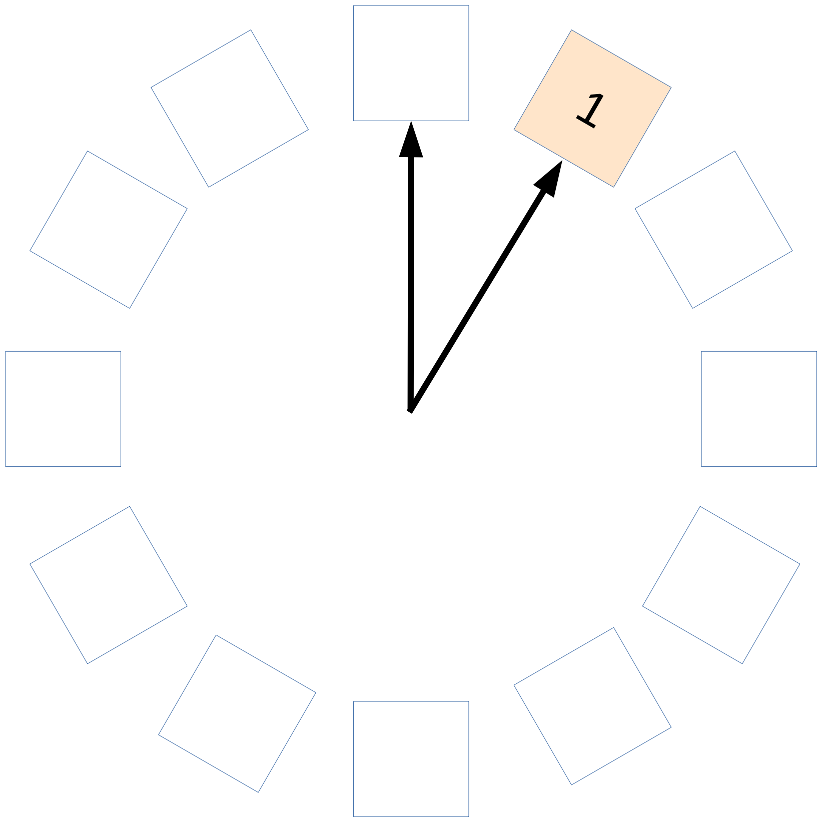
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



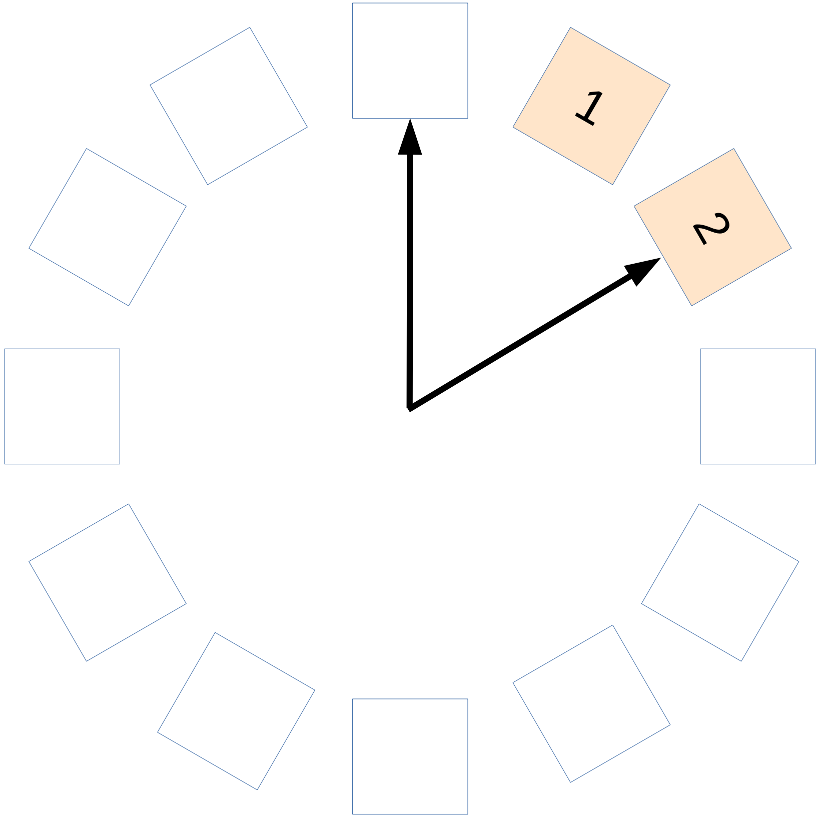
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



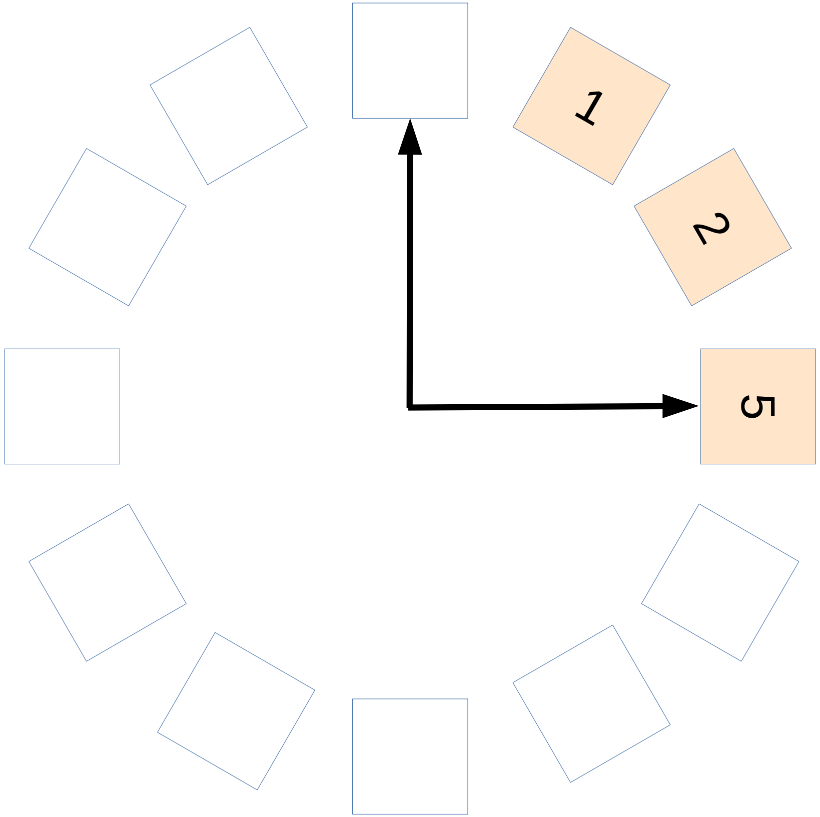
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



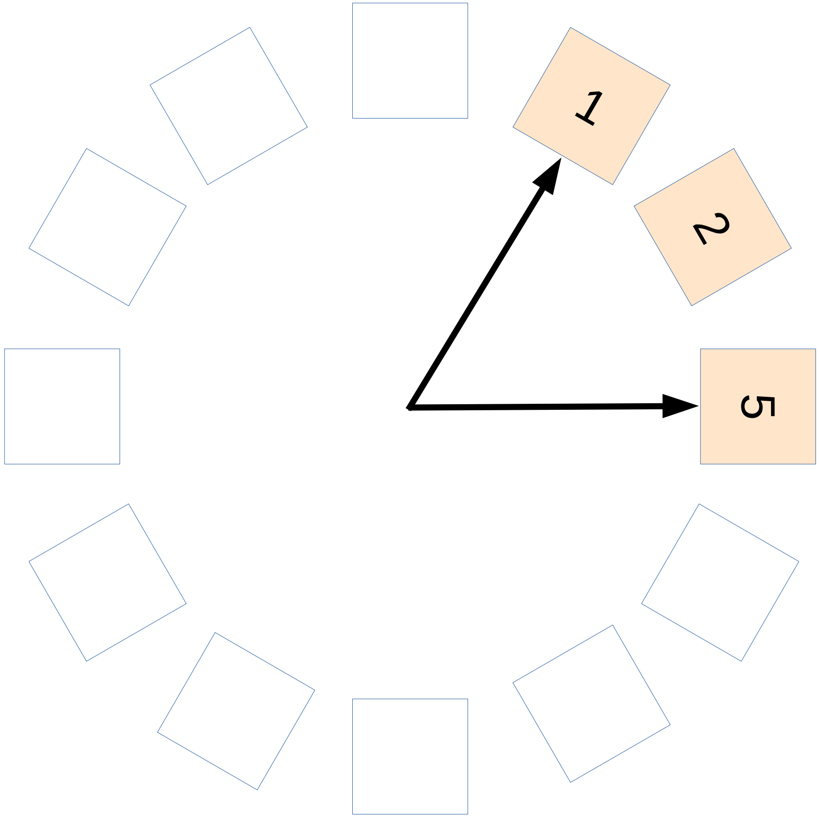
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



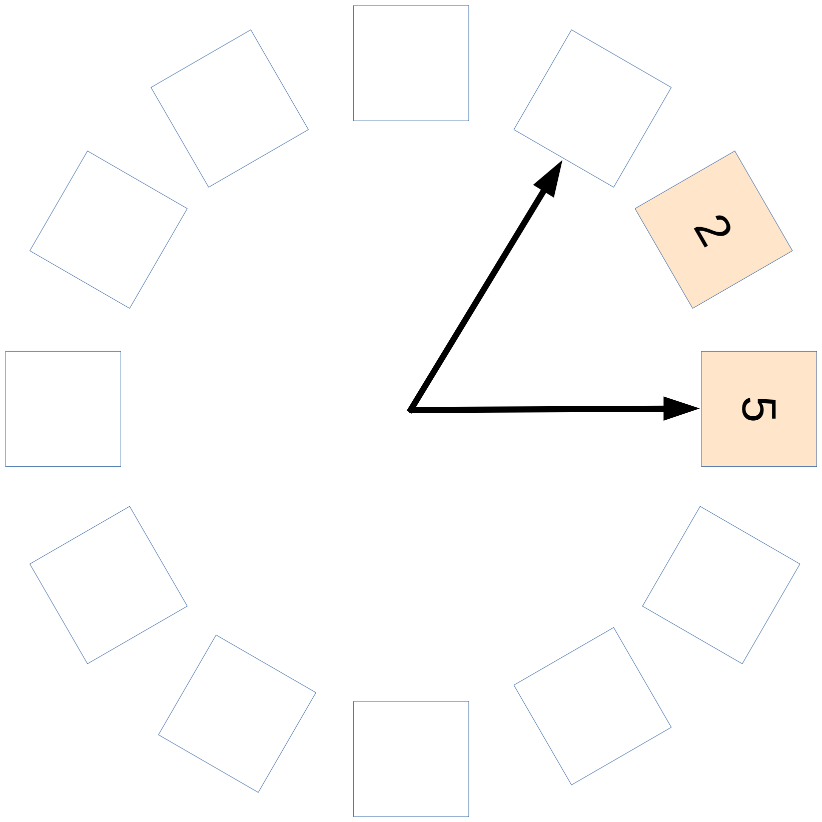
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



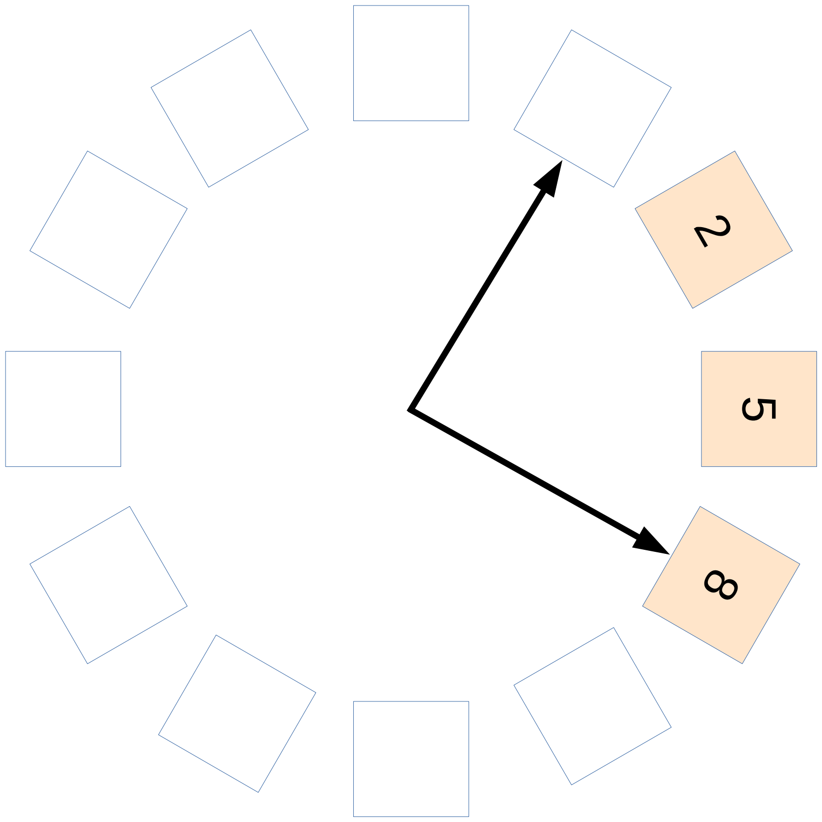
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



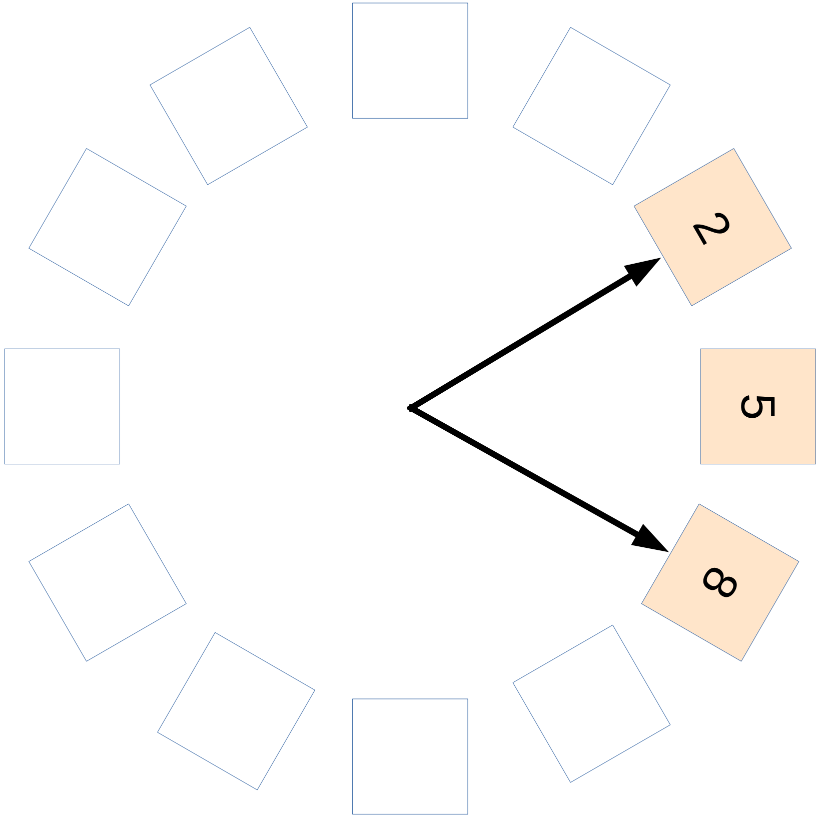
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



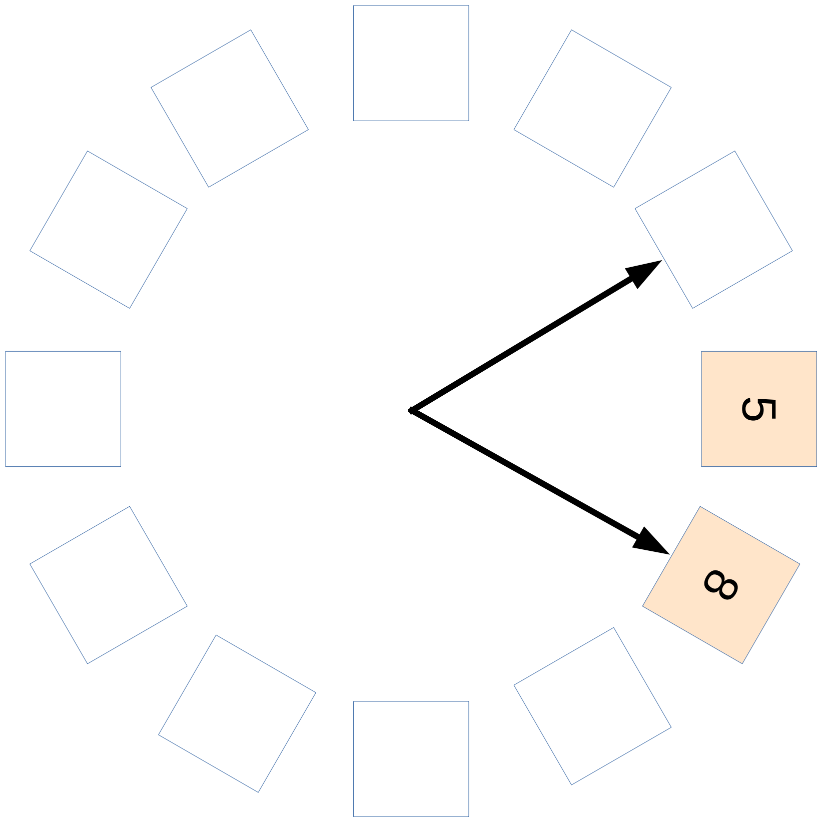
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



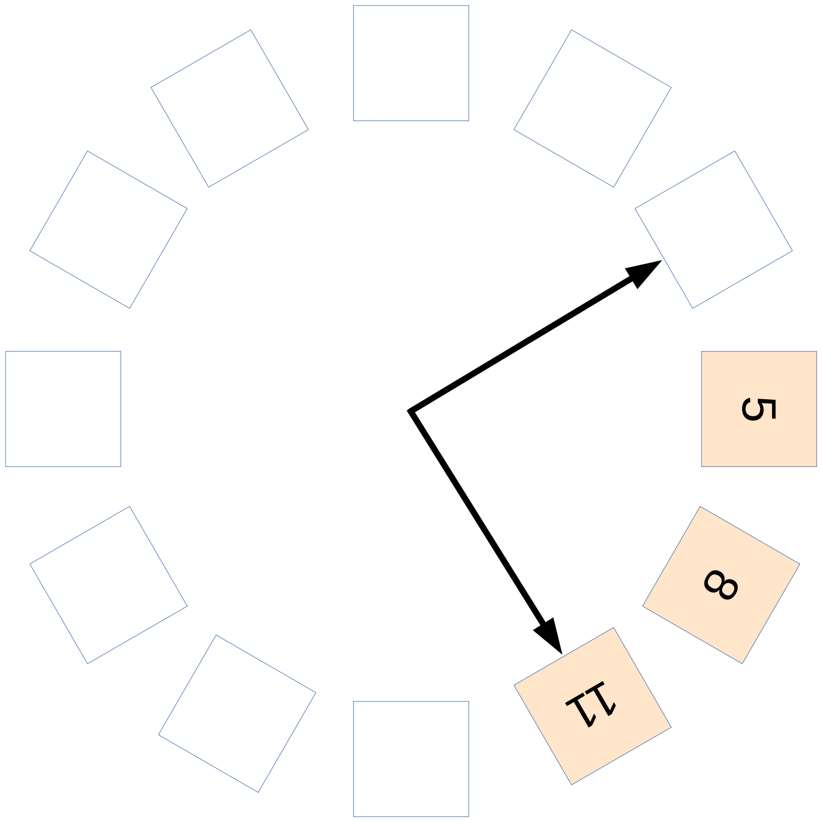
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



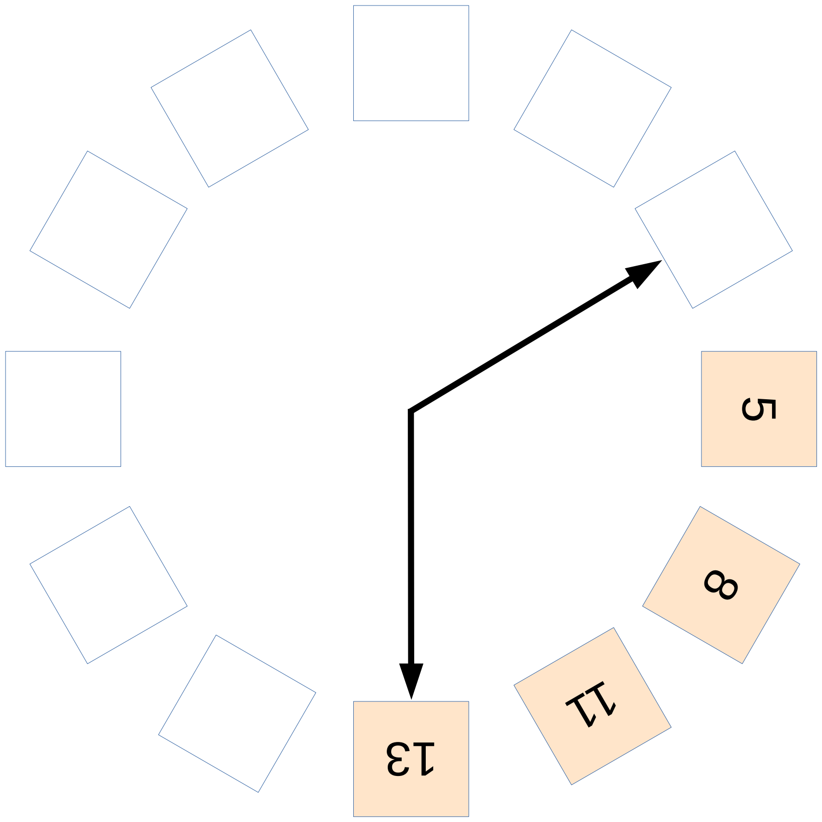
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



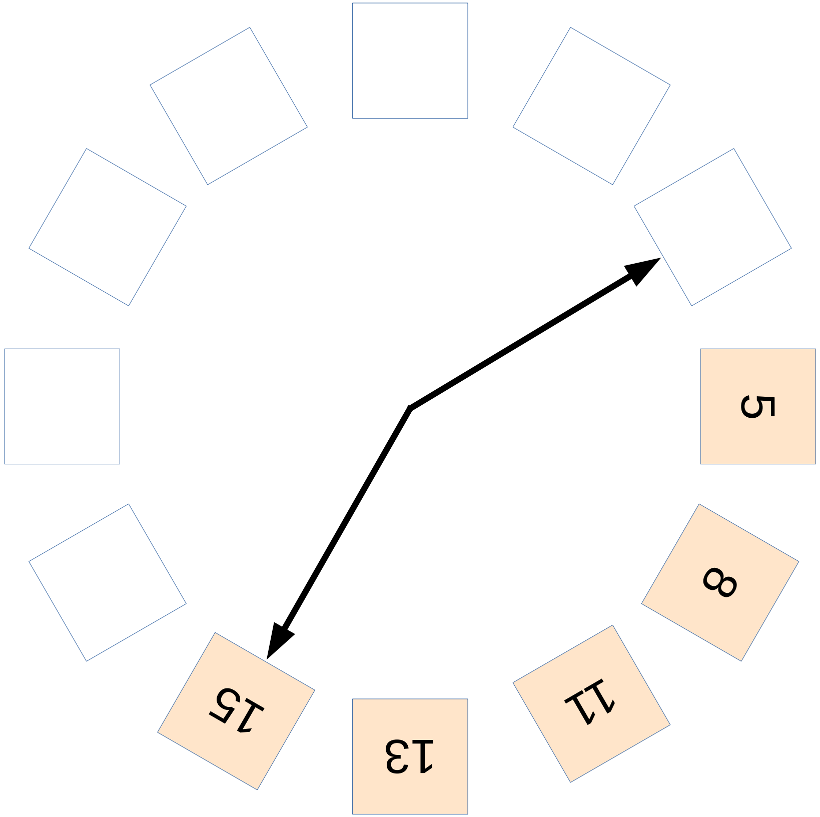
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



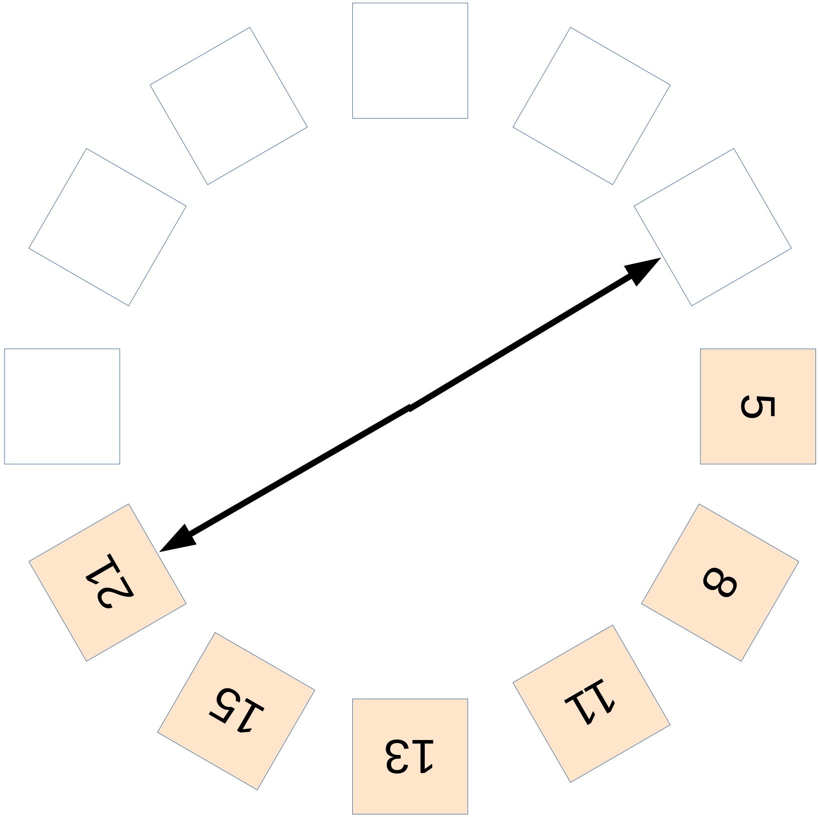
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



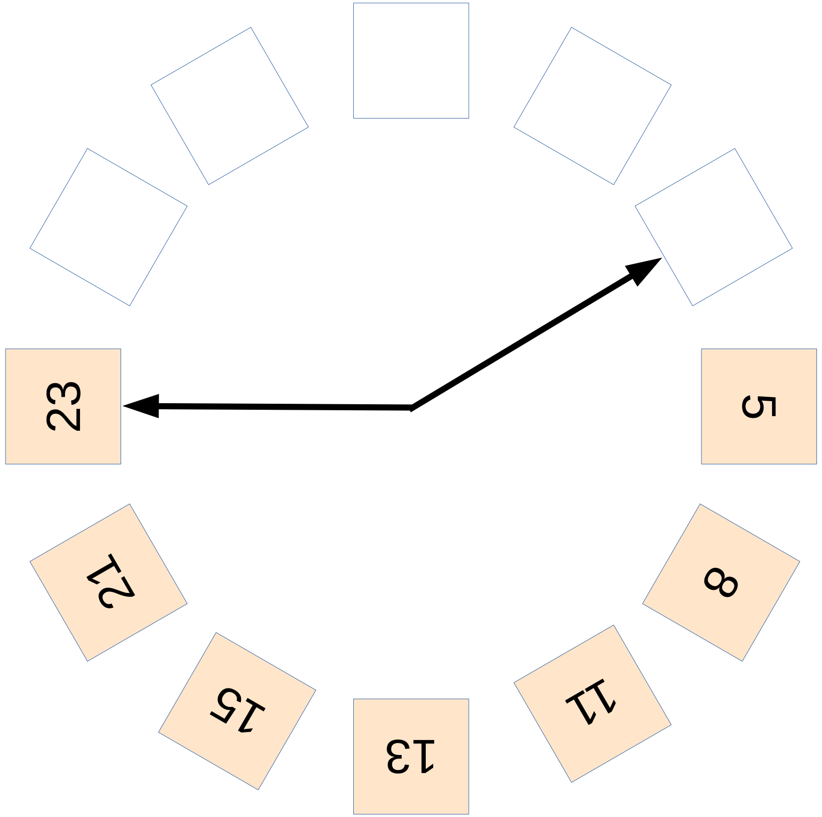
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



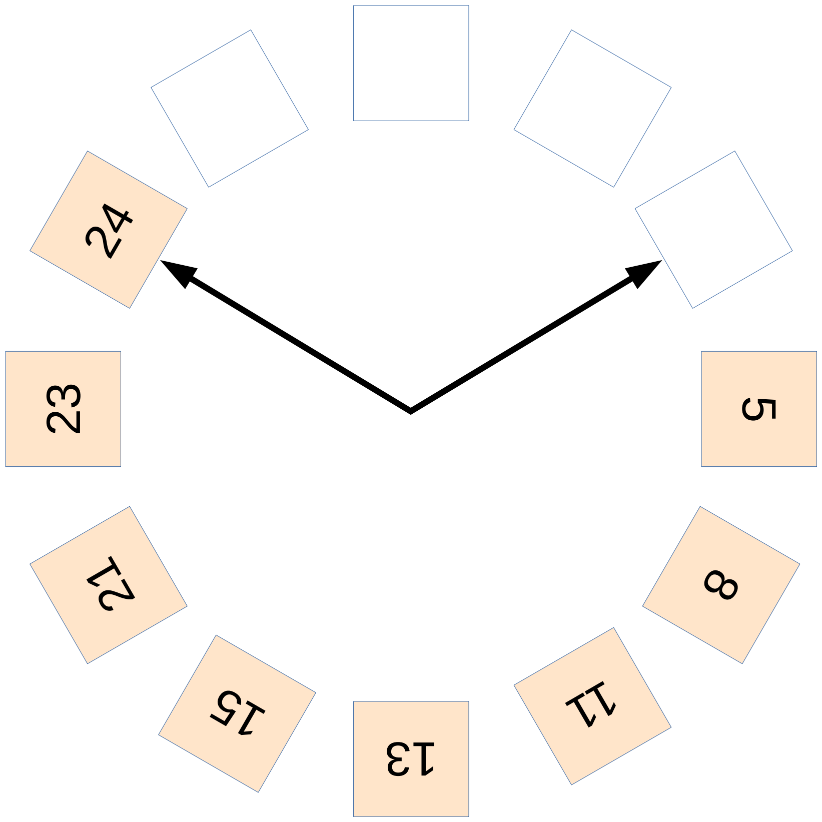
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



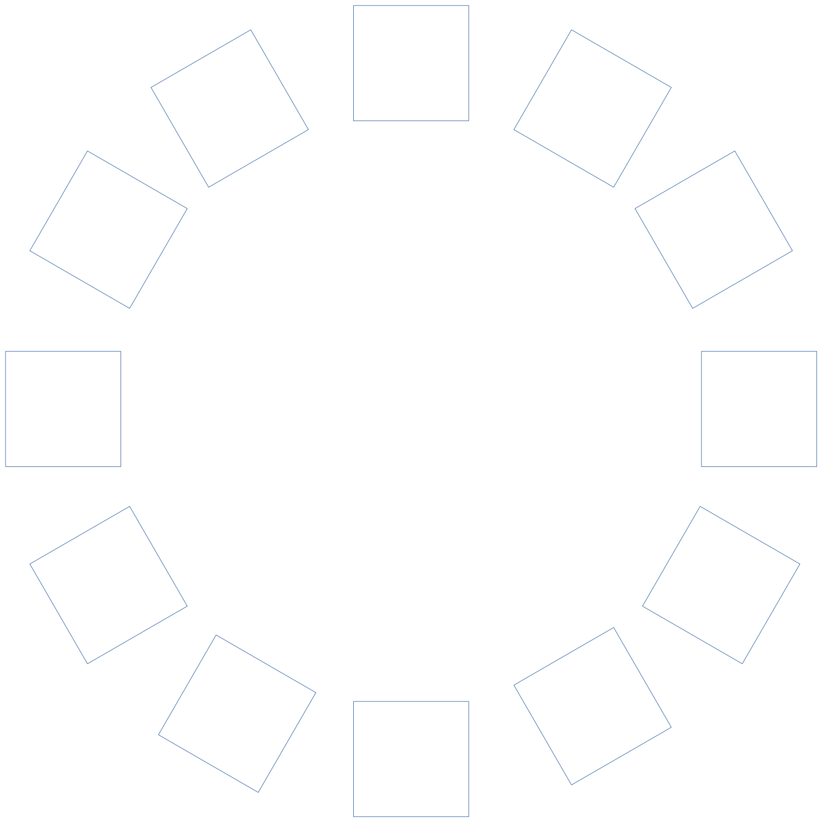
Ringbuffer example

```
ringbuffer <int,12> b;
b.push_back(1);
b.push_back(2);
b.push_back(5);
b.pop_front(); // 1
b.push_back(8);
b.pop_front(); // 2
b.push_back(11);
b.push_back(13);
b.push_back(15);
b.push_back(21);
b.push_back(23);
b.push_back(24);
```



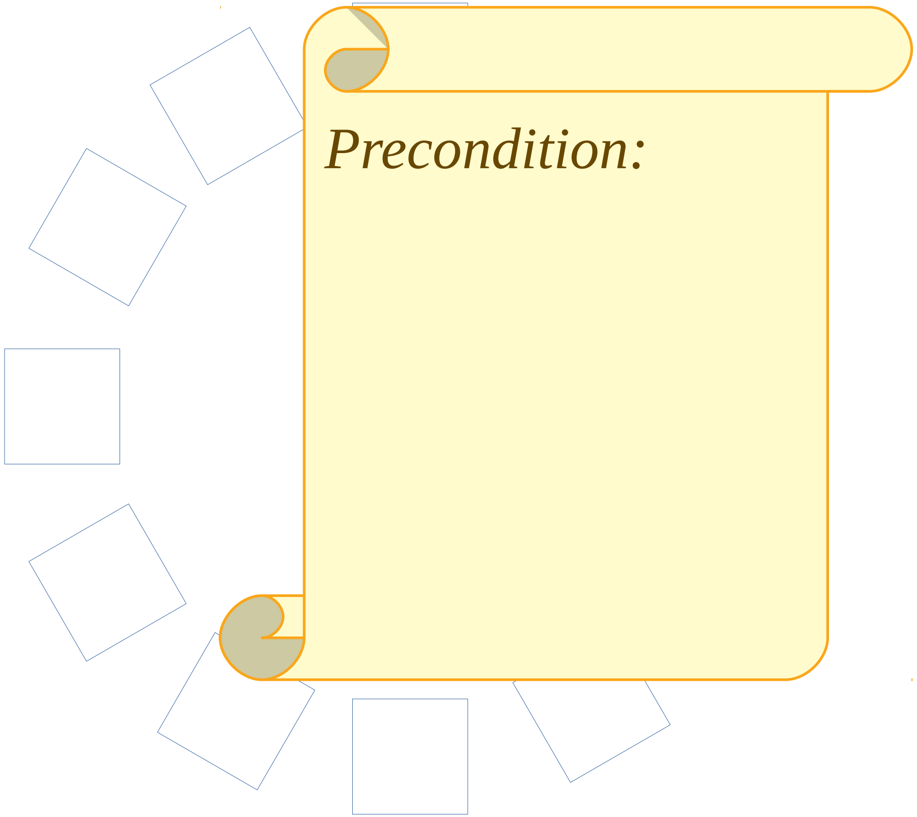
Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



Ringbuffer example

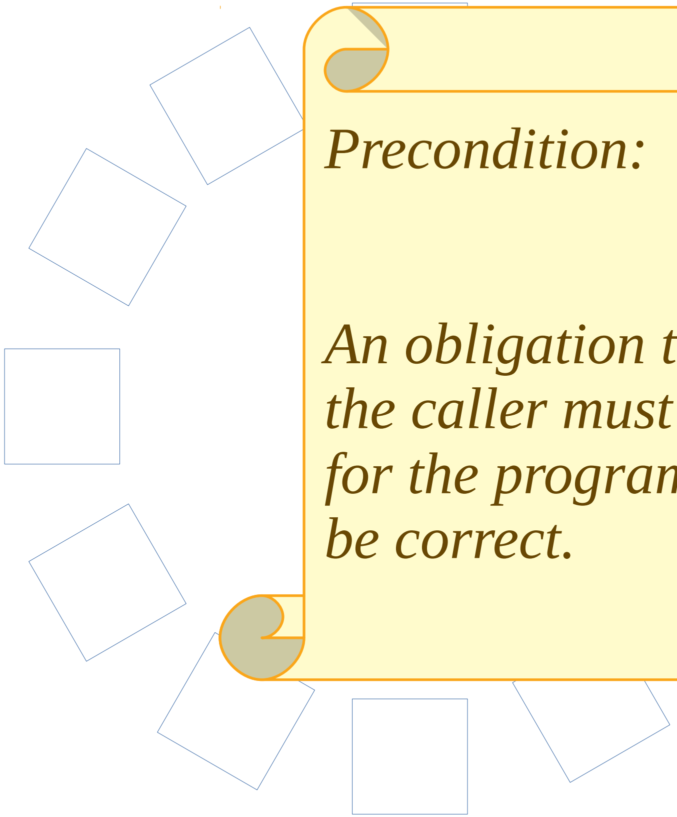
```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



Precondition:

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;



  void push_back(T);



  T pop_front(

};
```

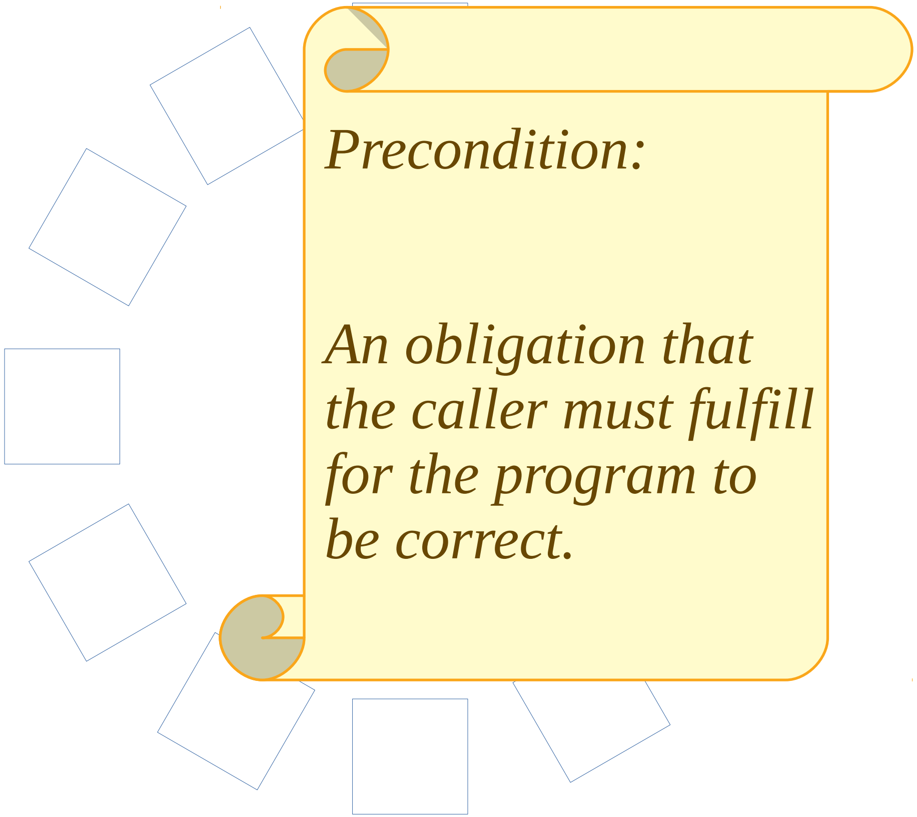
A precondition may refer to parameter values or the objects state, or both

Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



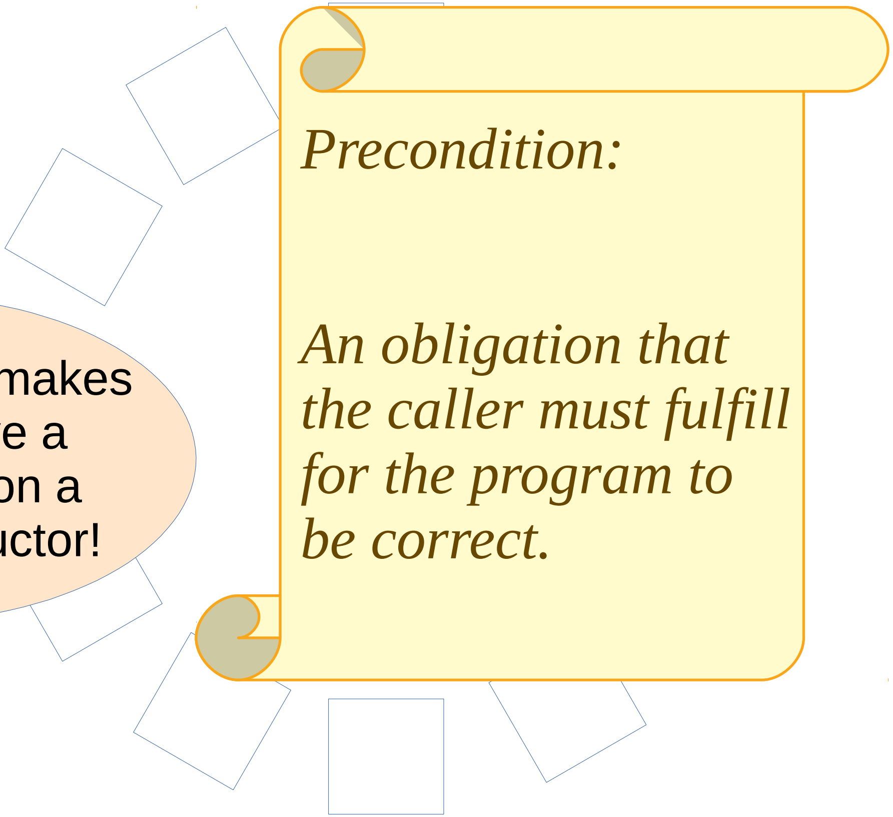
Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```

It almost never makes sense to have a precondition on a default constructor!

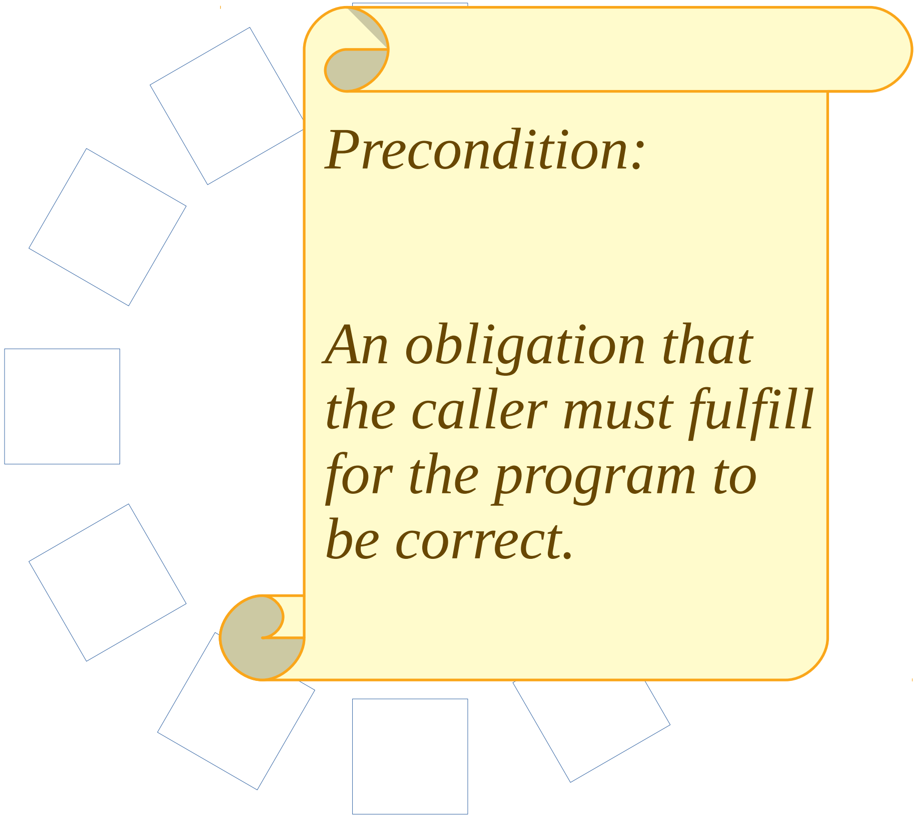


Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;



  void push_back(T);



  T pop_front();



};
```

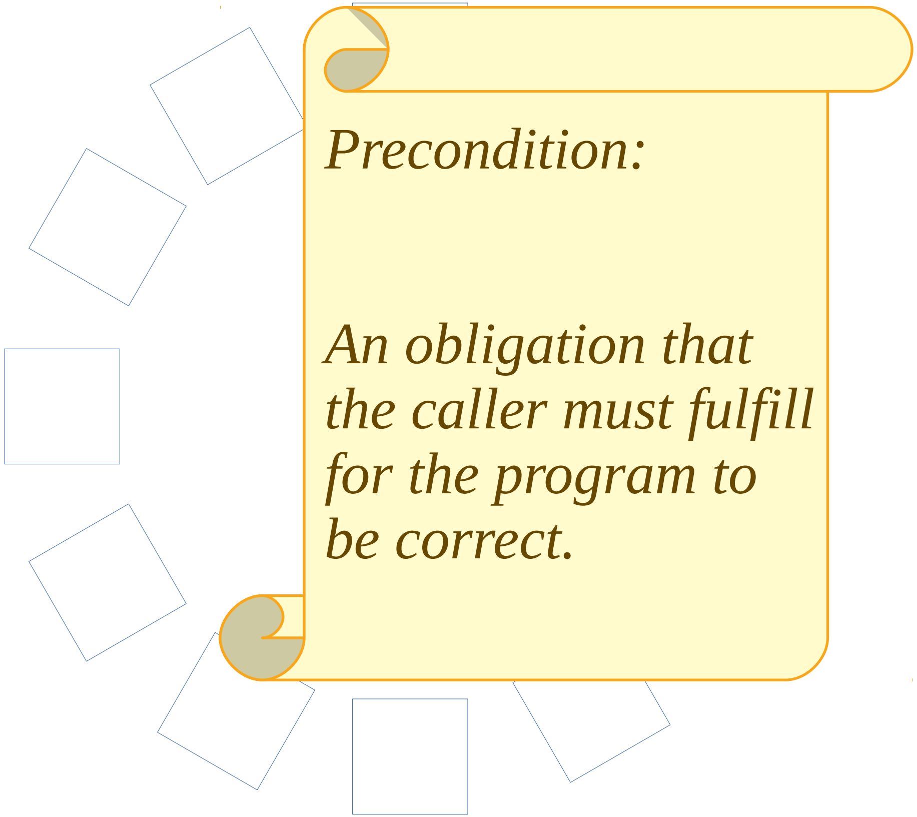
Functions that query the state of an object rarely has any preconditions.

Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);



  T pop_front();



};
```



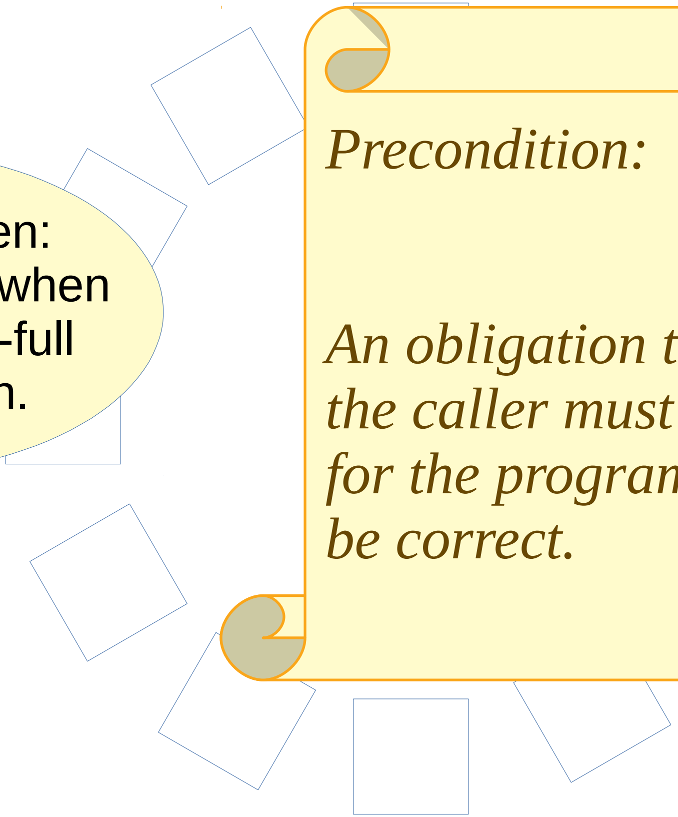
Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const

  void push_back(T);


  T pop_front();


};
```

Choose between:
Define behaviour when
full, or make not-full
a precondition.

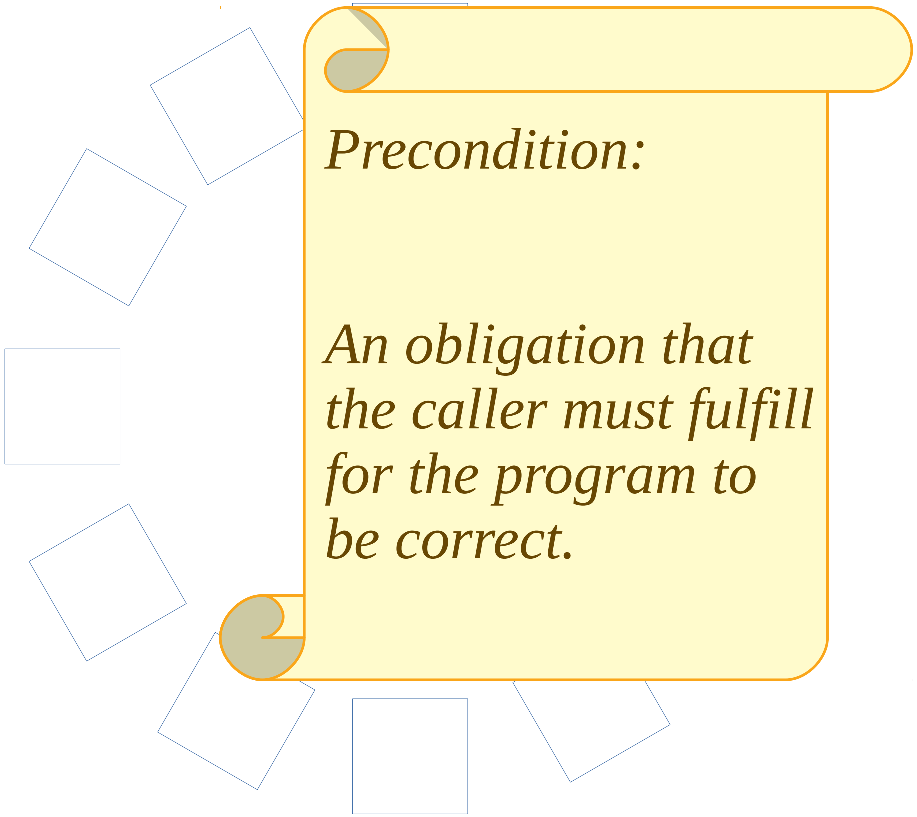


Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();



};
```

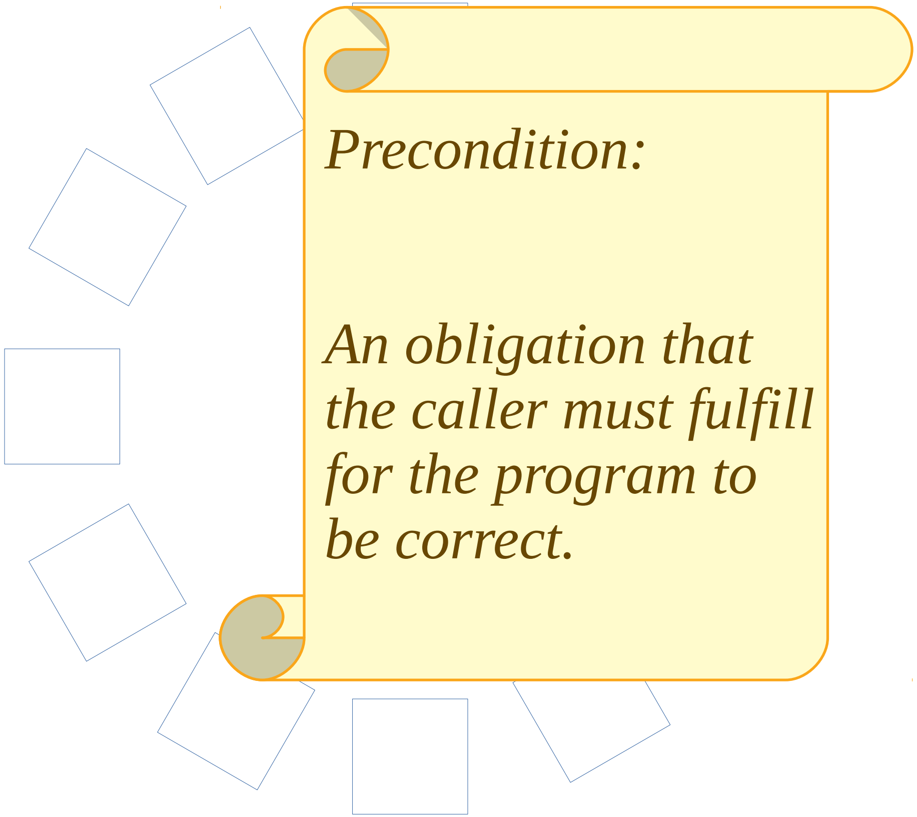


Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();



};
```



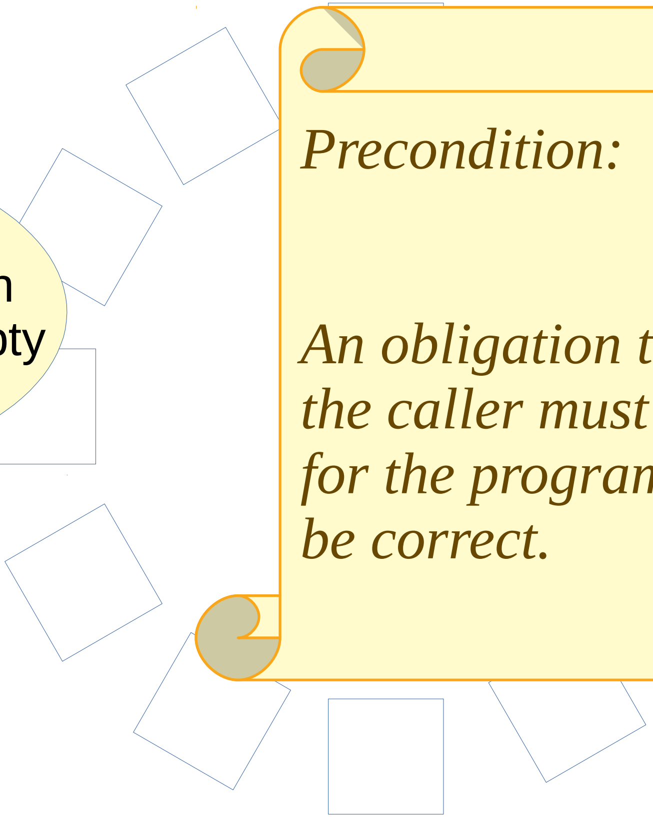
Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() co



  void push_back(
  // requires:    e() < N


  T pop_front();



};
```

Choose between:
Define behaviour when
empty, or make not-empty
a precondition.

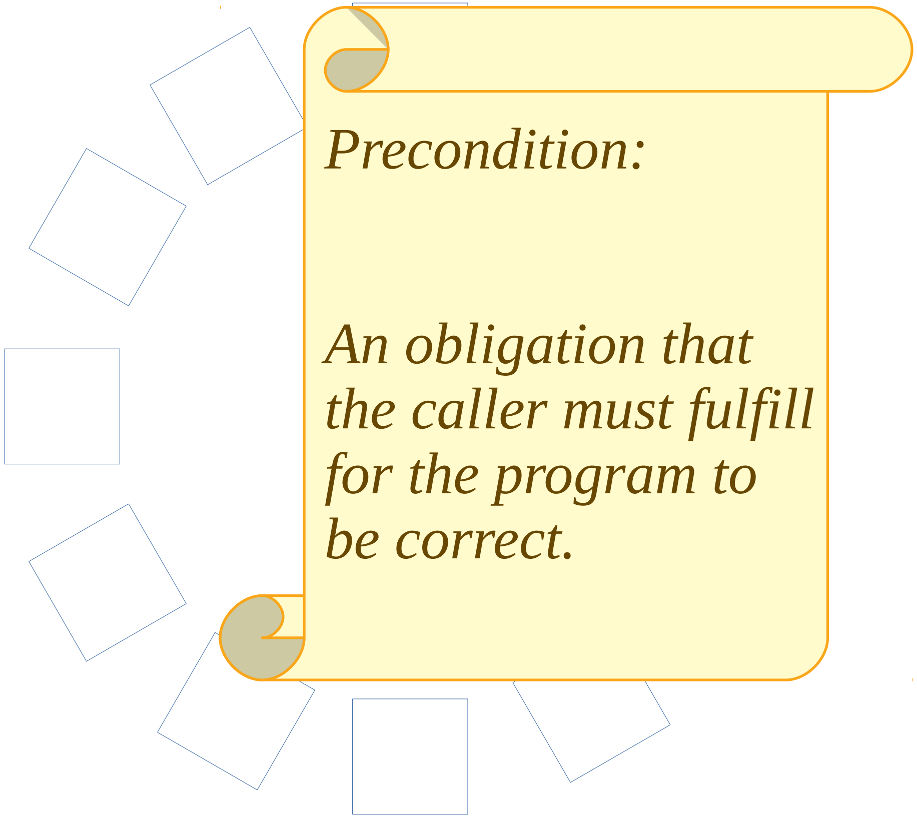


Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```

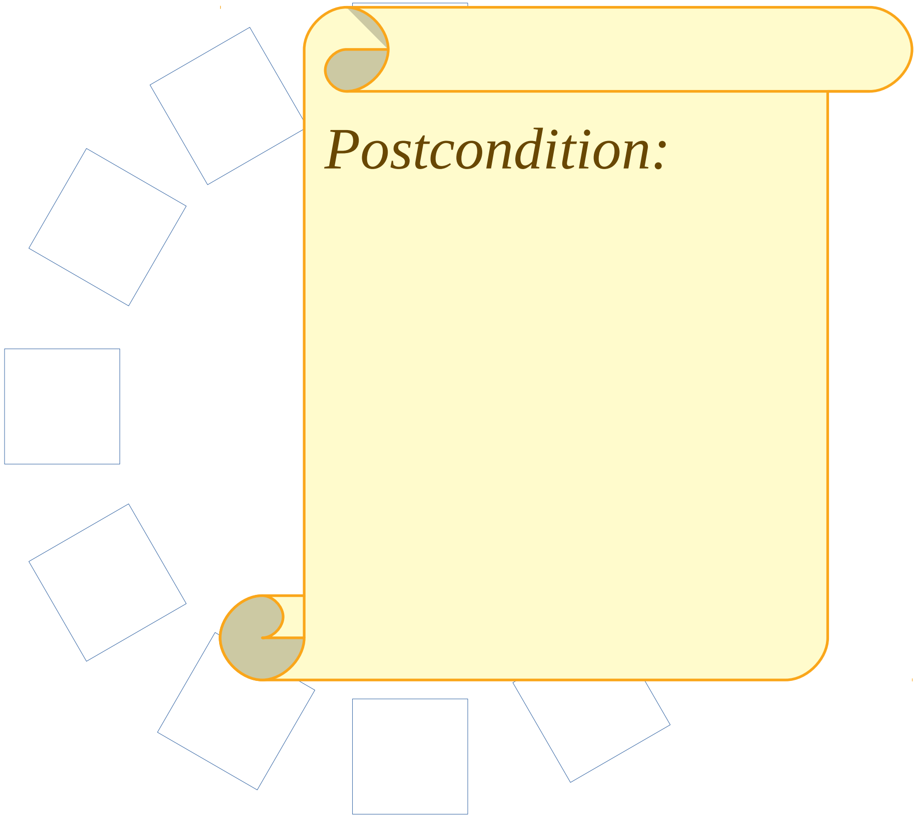


Precondition:

An obligation that the caller must fulfill for the program to be correct.

Ringbuffer example

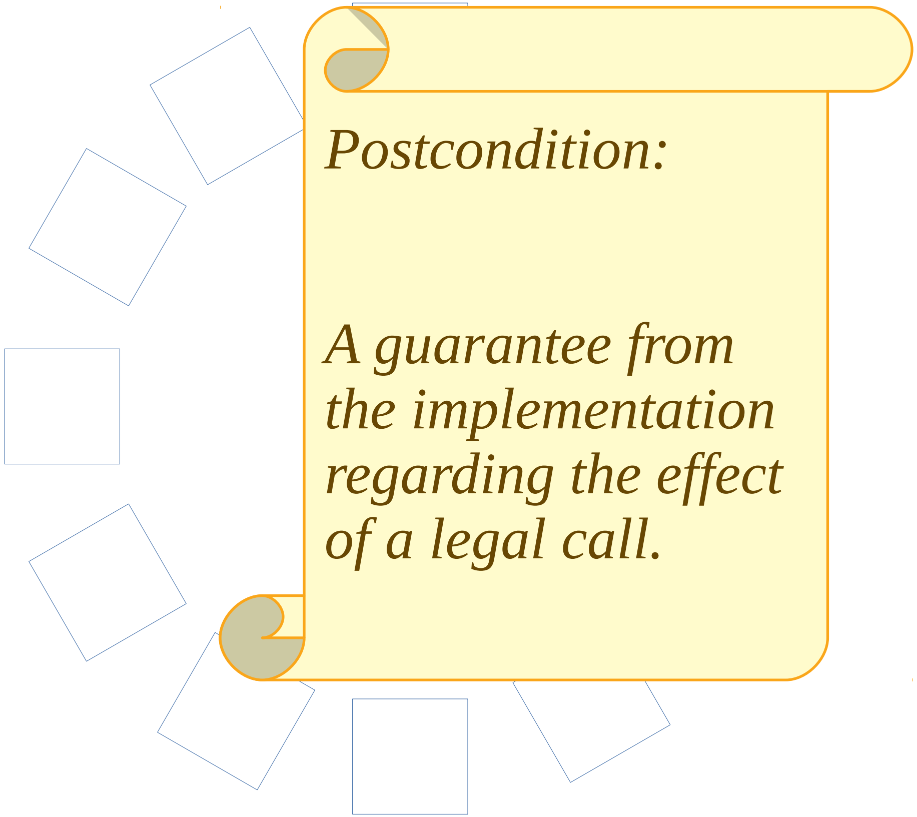
```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;





  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```



Postcondition:

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```



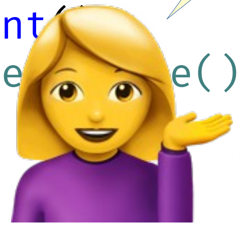
Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;



  void push_back(T);
  // requires: size()


  T pop_front(
  // require        e() > 0


};
```

A postcondition may refer to return value or the objects state, or both, sometimes dependent on parameter values

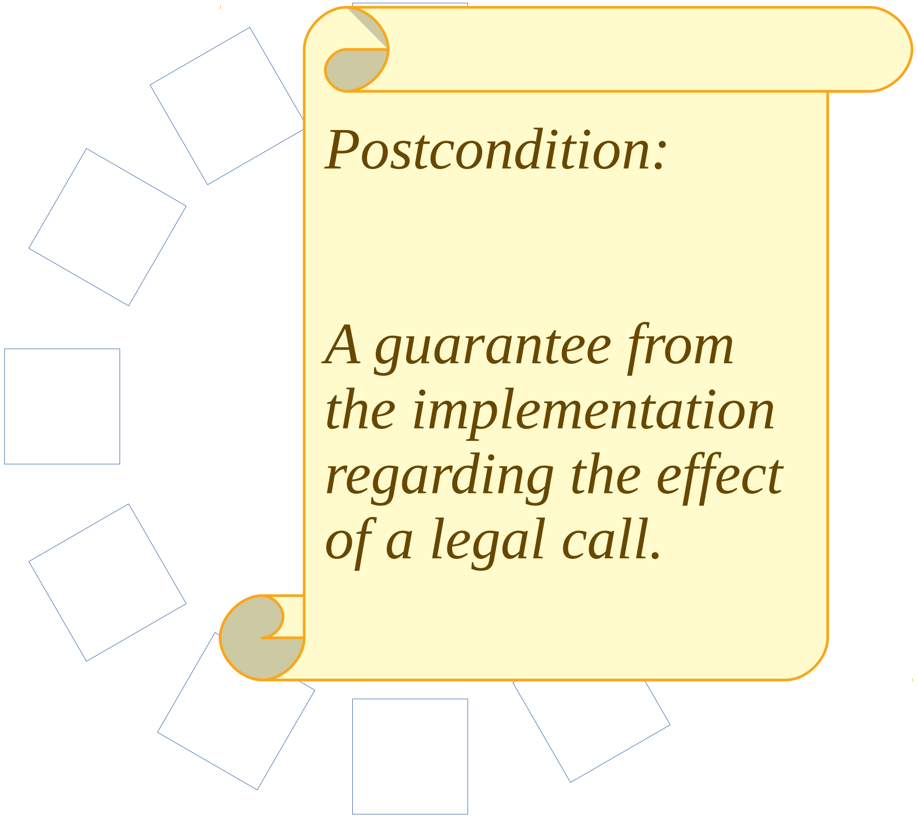


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();

  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```

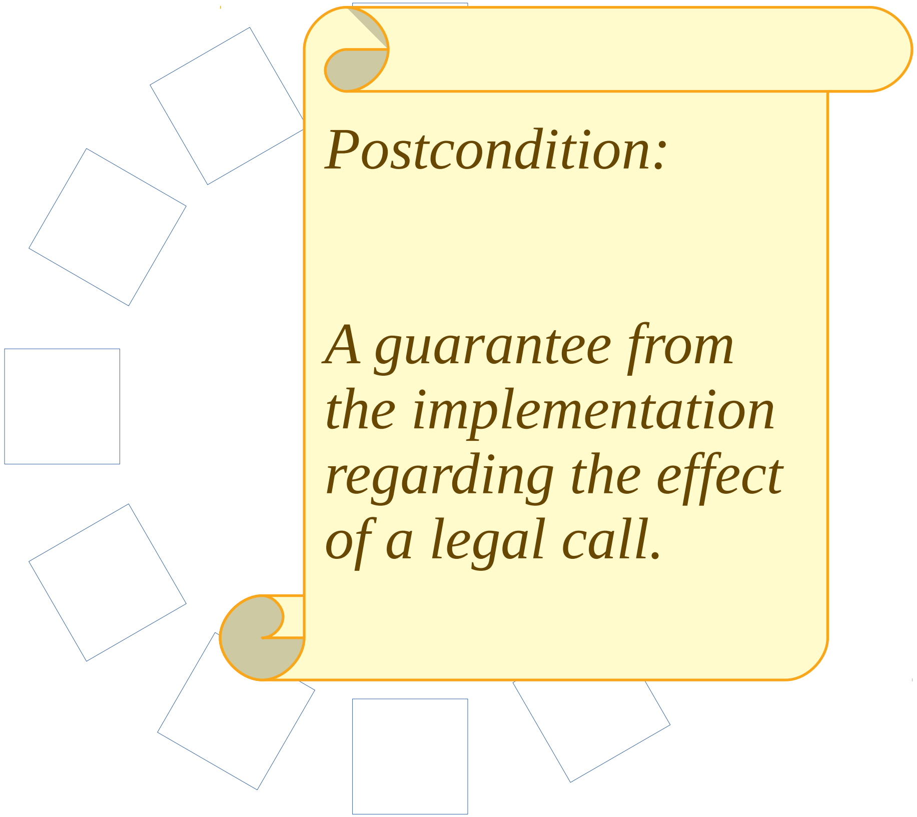


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```

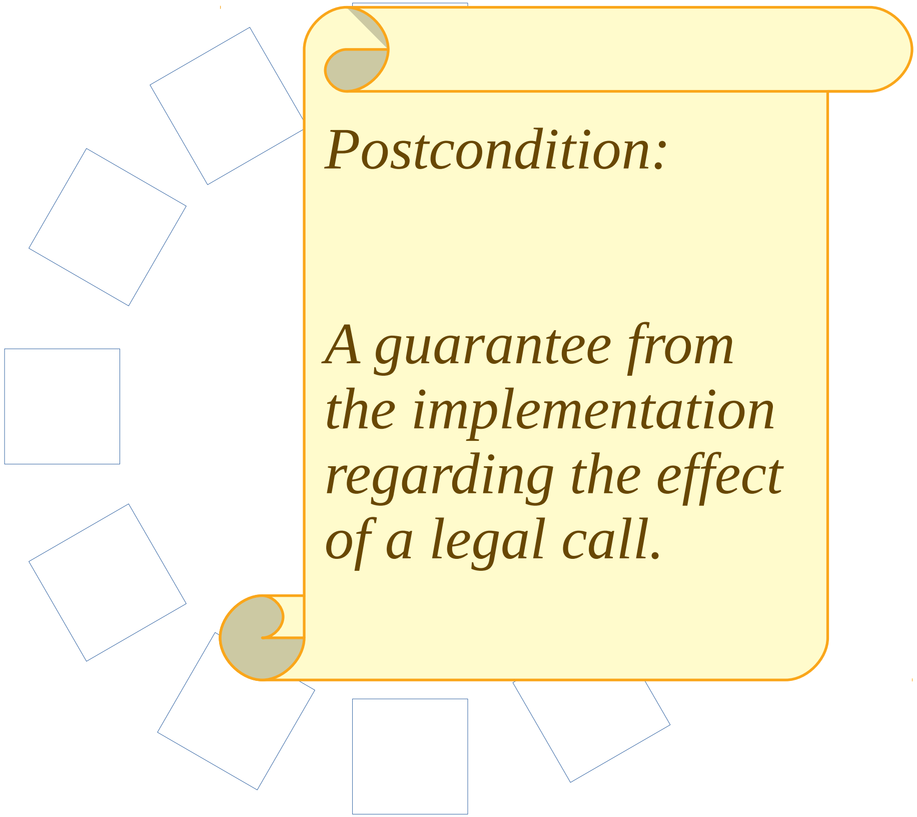


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;



  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```

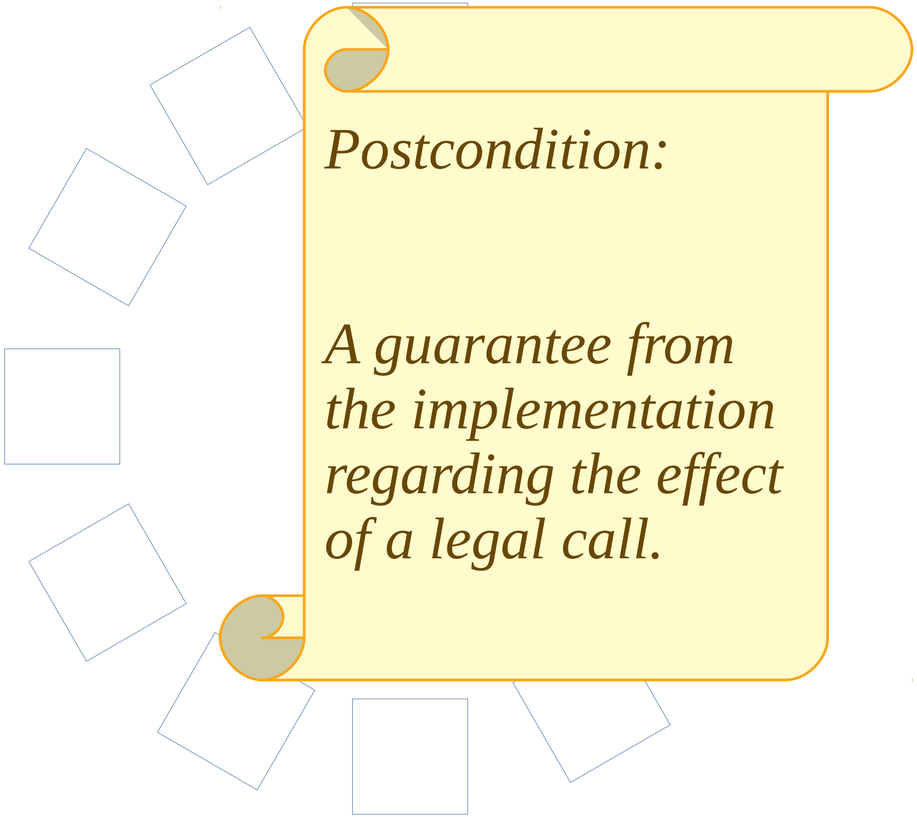


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;




  void push_back(T);
  // requires: size() < N


  T pop_front();
  // requires: size() > 0


};
```

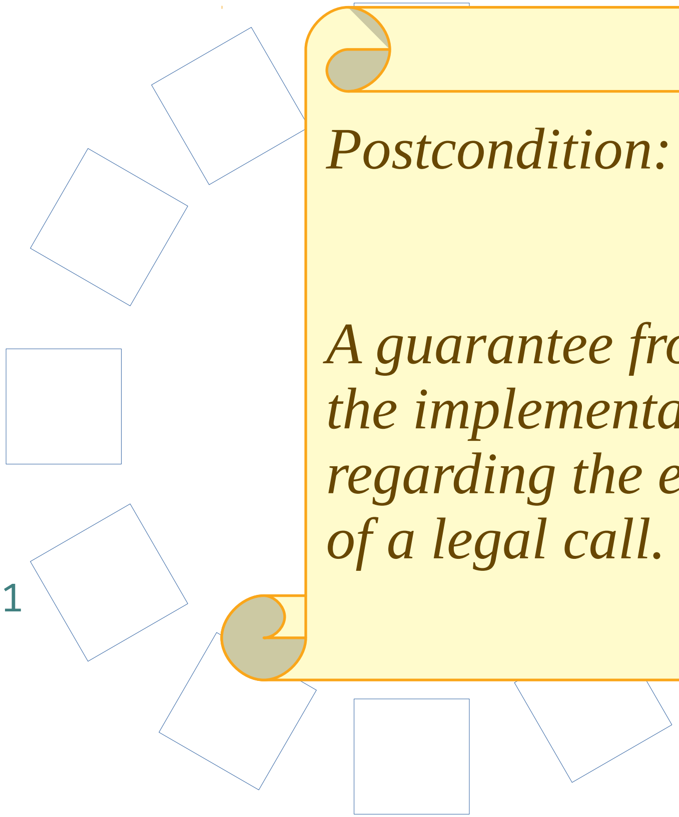


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;



  void push_back(T);
  // requires: size() < N
  // ensures: size() == old size()+1

  T pop_front();
  // requires: size() > 0


};
```

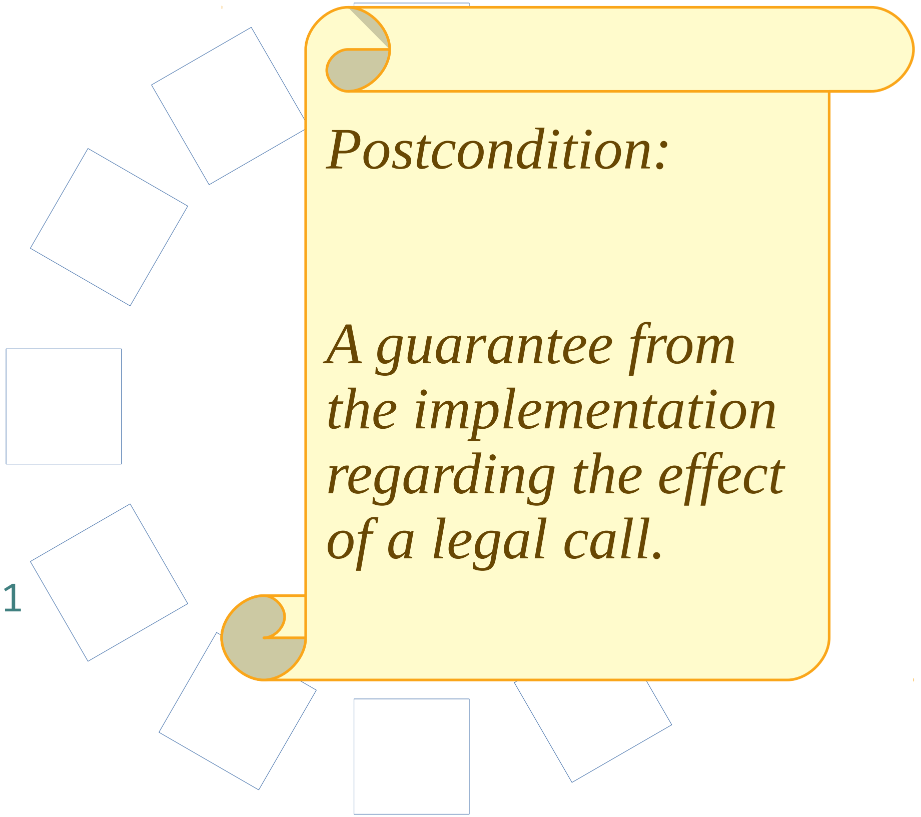


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;




  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1

  T pop_front();
  // requires: size() > 0


};
```

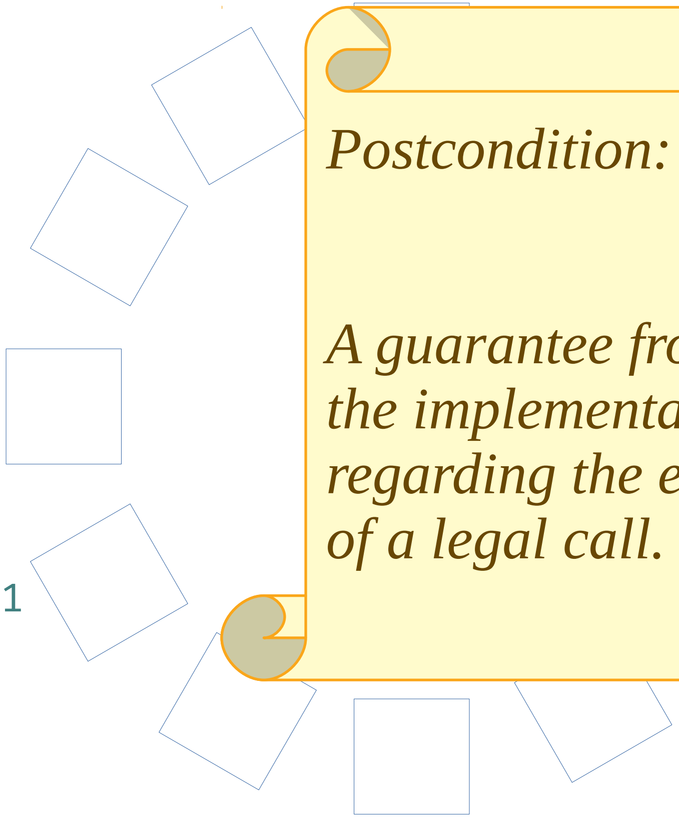


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;



  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0


};
```

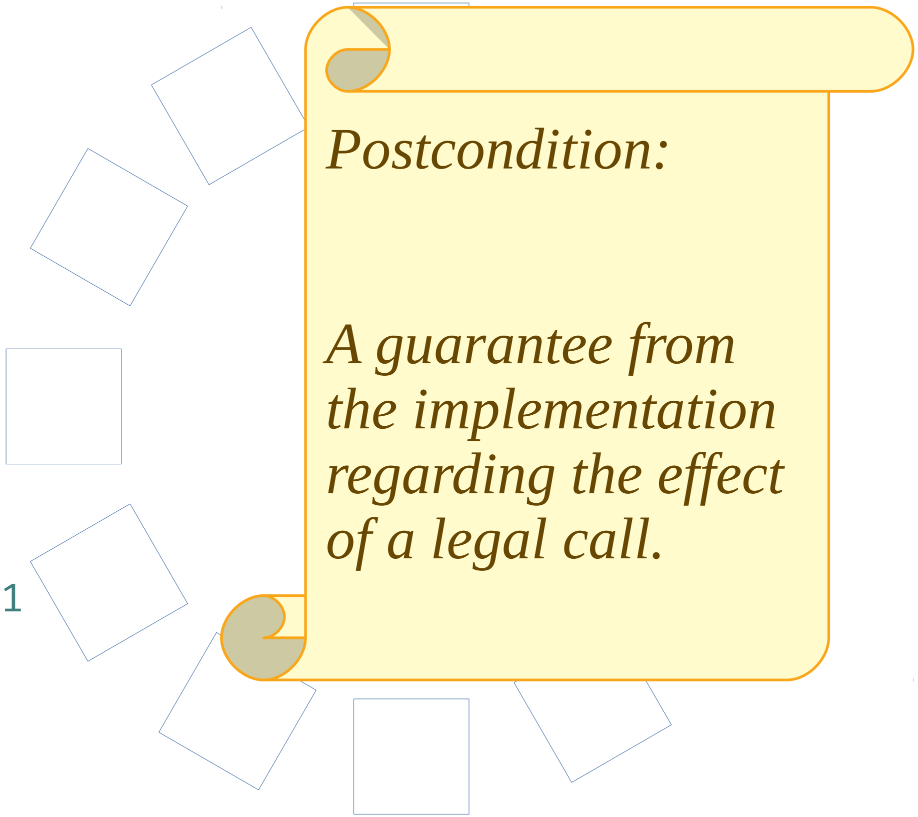


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0


  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0


};
```



Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures:
  int size() co
  const T& back(
  // requires: size() > 0


  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()
  //          back() == t
  T pop_front();
  // requires: size() > 0


};
```

What if an exception is thrown?

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() co
  // requires: size(


  void push_back(T t);
  // requires: size() <
  // ensures: size() == old size()+1
  //             back() == t
  T pop_front(
  // require     e() > 0


};
```

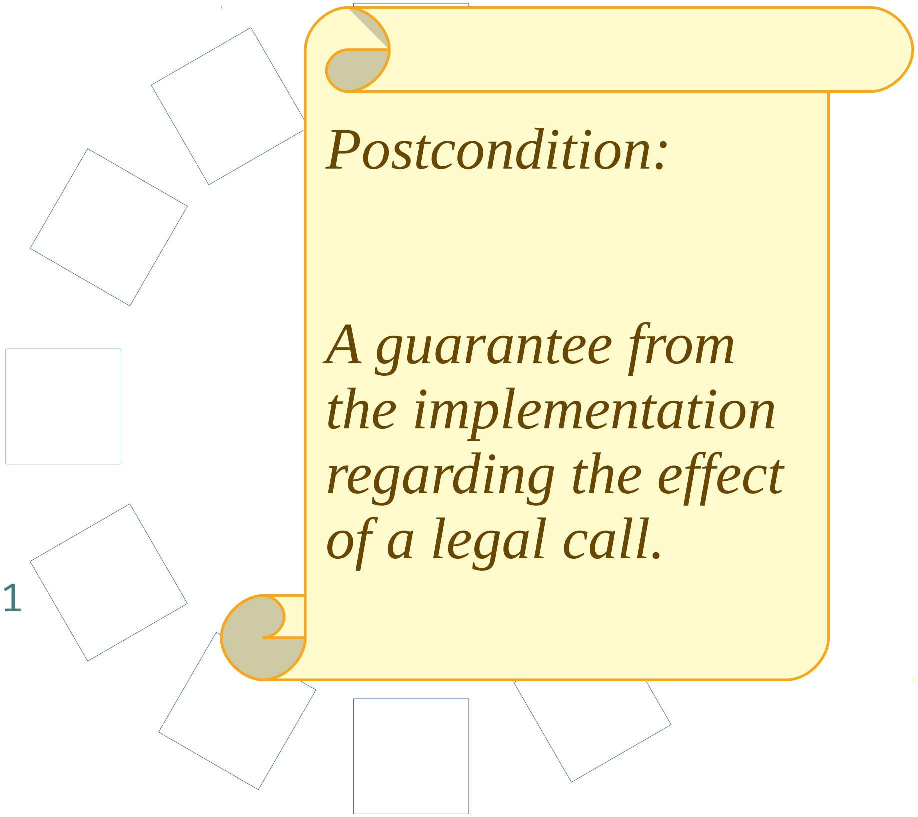
Postconditions handles return. If an exception is thrown, there is no post condition.

Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0


  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //                back() == t
  T pop_front();
  // requires: size() > 0


};
```

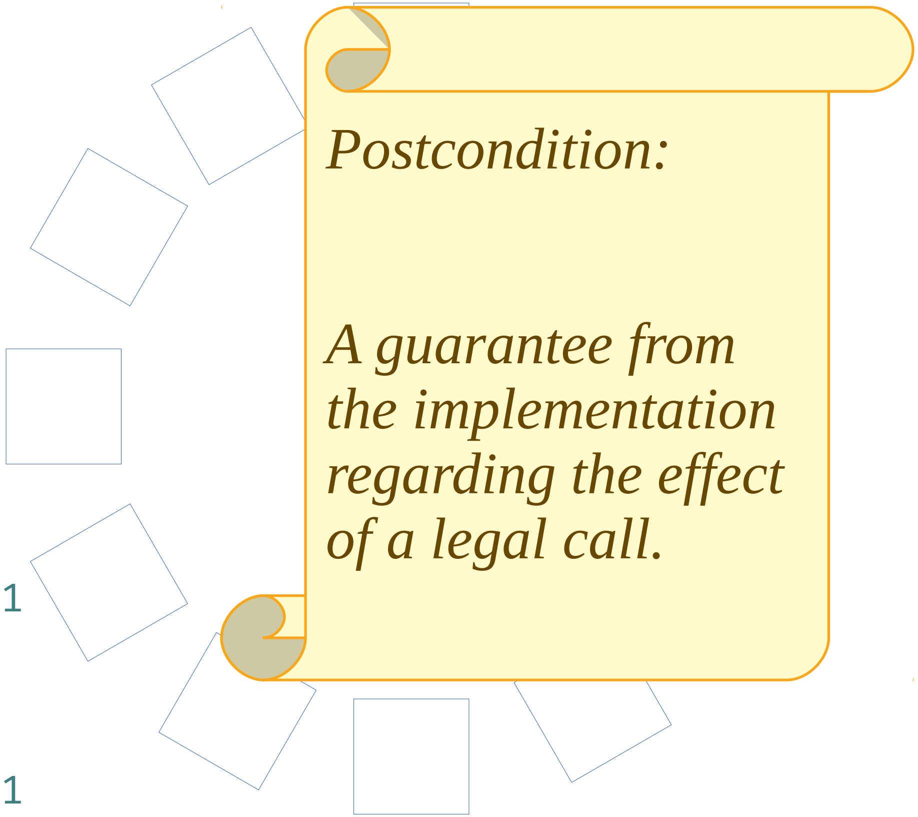


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0


  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //                back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1

};
```

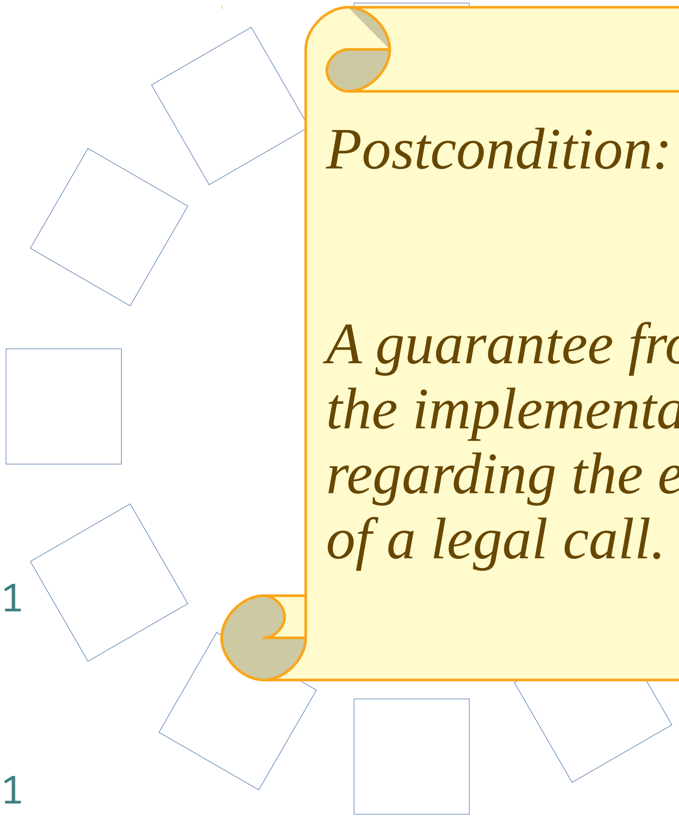


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;

  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

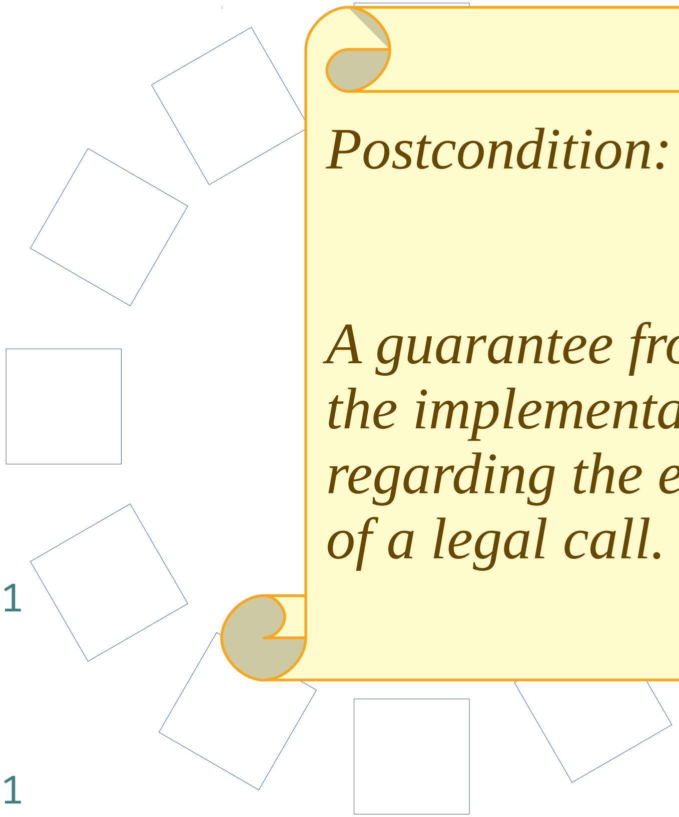


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



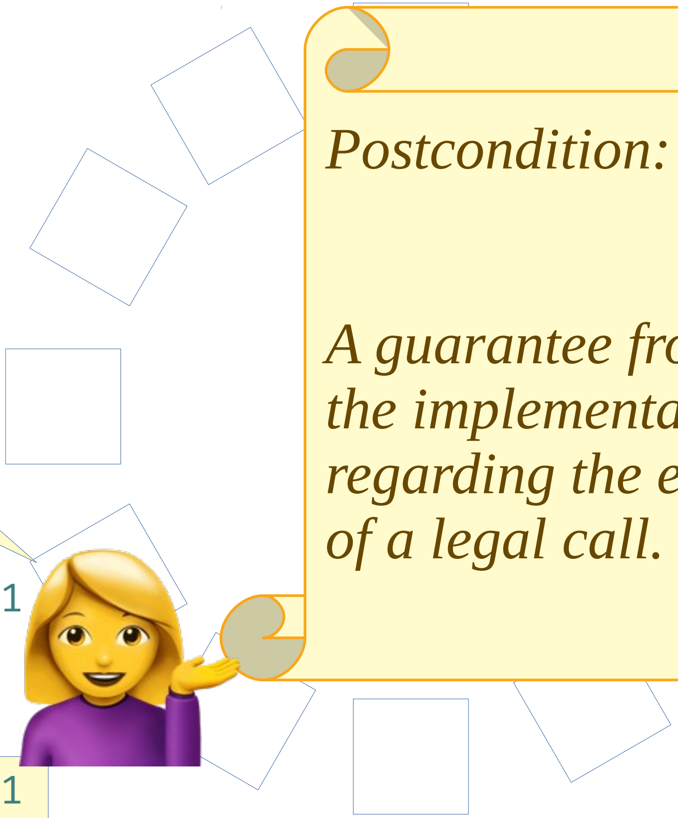
Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  //
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

It does not make sense to try and express the returned value from the history of pushes and pops as a post condition.

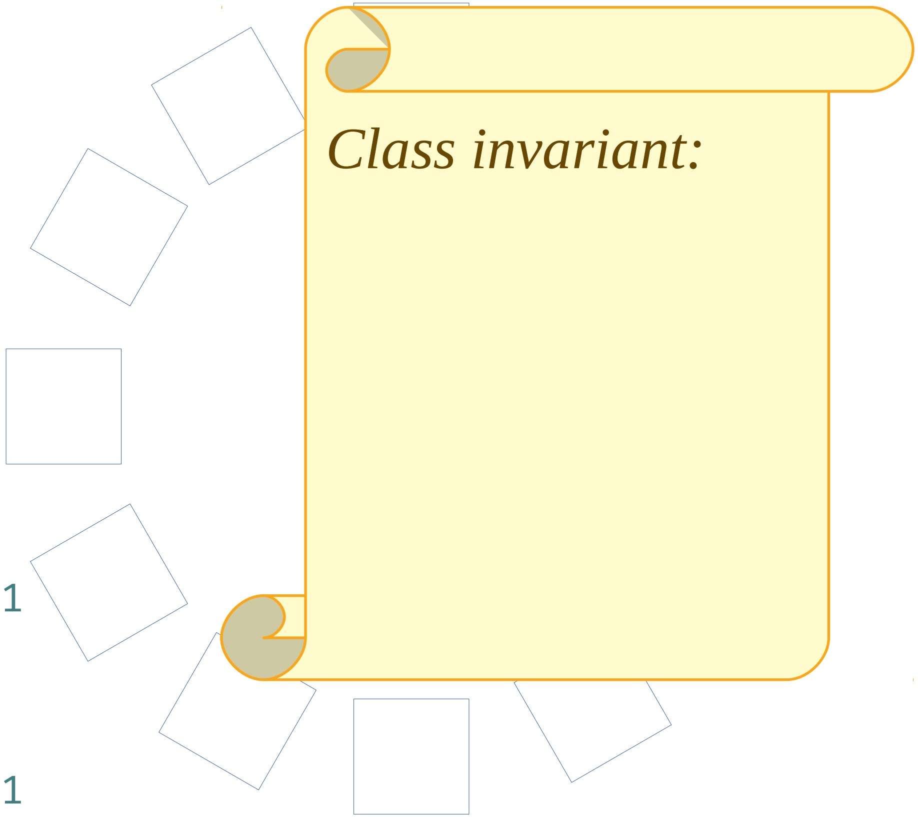


Postcondition:

A guarantee from the implementation regarding the effect of a legal call.

Ringbuffer example

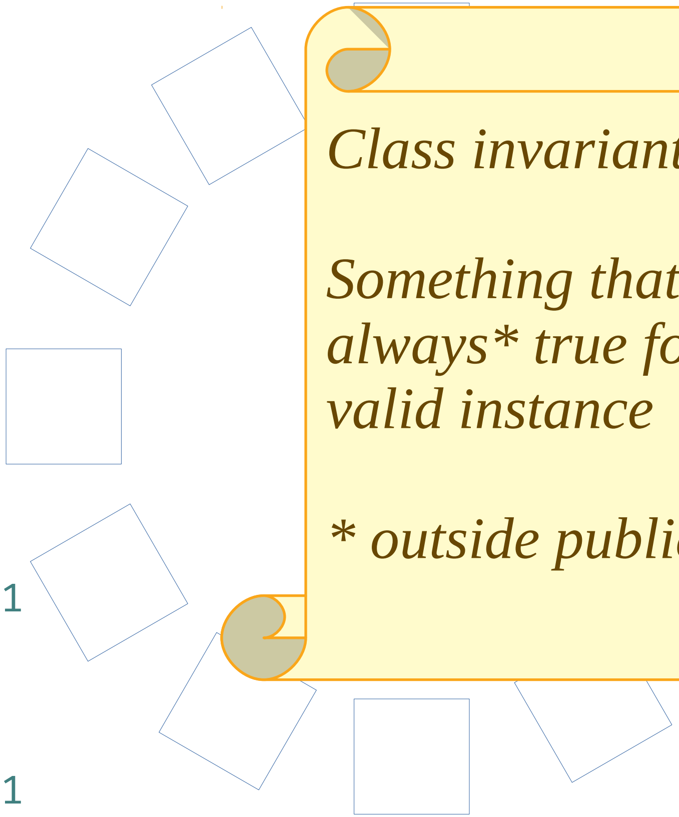
```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



Class invariant:

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



Class invariant:

Something that is always true for a valid instance*

** outside public API*

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:

  ringbuffer();
  // ensures: size() ==
  int size() const;
  const T& back() con
  // ensures: size() >
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() <
  // ensures: size()   = old size()+1
  //              back()  == t
  T pop_front(
  // require     e() > 0
  // ensures     () == old size()-1
  //              n == old front();
};
```

A class invariant always refers to state, and must be true even when exceptions are thrown.

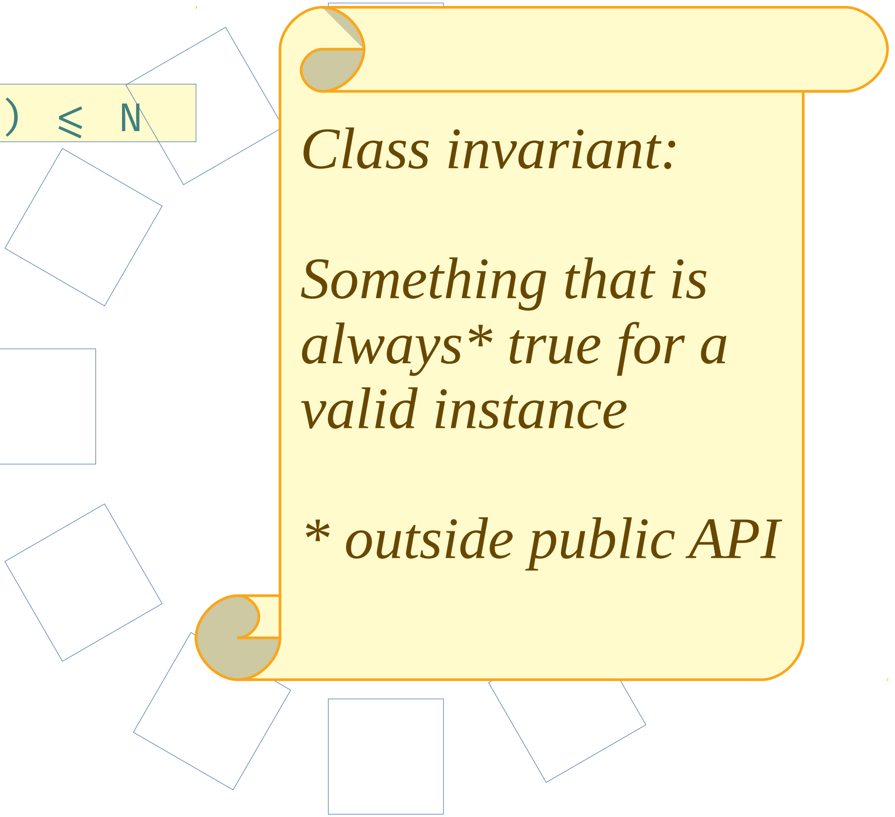
Class invariant:

Something that is always true for a valid instance*

** outside public API*

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //             back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //             return == old front();
};
```



Class invariant:

Something that is always true for a valid instance*

** outside public API*

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() =
  int size() const;
  const T& back() con
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

What about a moved-from object?

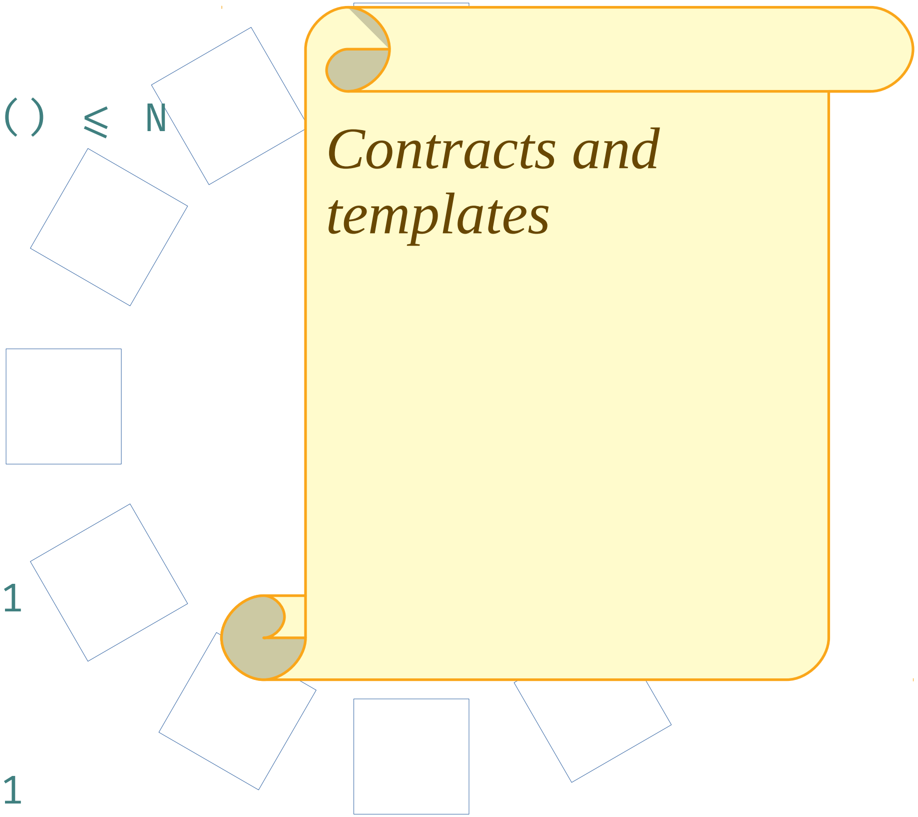
Class invariant:

Something that is always true for a valid instance*

** outside public API*

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

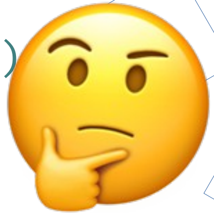


Contracts and templates

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() =
  int size() const;
  const T& back() con
  // ensures: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

What about specializations?



Contracts and templates

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  virtual int size() const = 0;
  virtual const T& back() const = 0;
  // requires: size() > 0
  virtual const T& front() const = 0;
  // requires: size() > 0
  virtual void push_back(T t) = 0;
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  virtual T pop_front() = 0;
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Contracts and inheritance:

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  virtual int size() const = 0;
  virtual const T& back() const = 0;
  // requires: size() > 0
  virtual const T& front() const = 0;
  // requires: size() > 0
  virtual void push_back(T t) = 0;
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  virtual T pop_front() = 0;
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Contracts and inheritance:

A subcontractor may have more relaxed pre-conditions

Ringbuffer example

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  virtual int size() const = 0;
  virtual const T& back() const = 0;
  // requires: size() > 0
  virtual const T& front() const = 0;
  // requires: size() > 0
  virtual void push_back(T t) = 0;
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  virtual T pop_front() = 0;
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Contracts and inheritance:

A subcontractor may have more relaxed pre-conditions

and stricter post-conditions

Why bother?

Why bother?

1) It can make interfaces much clearer

Why bother?

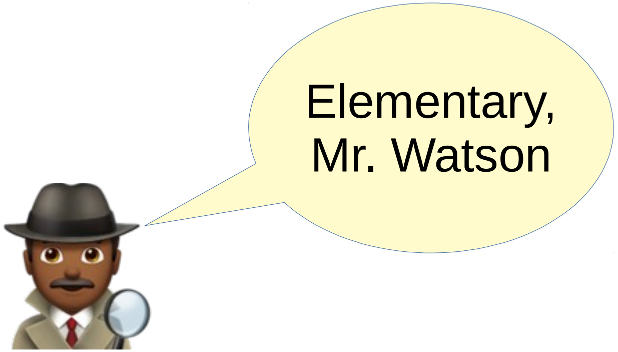
1) It can make interfaces much clearer

2) It can make debugging much easier

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

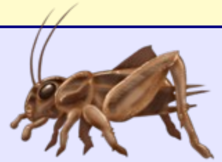


		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition		
	postcondition		
	invariant		

Who dunnit?

		guilty	
		client	implementation
violation	precondition	🦗	
	postcondition		🦗
	invariant		🦗

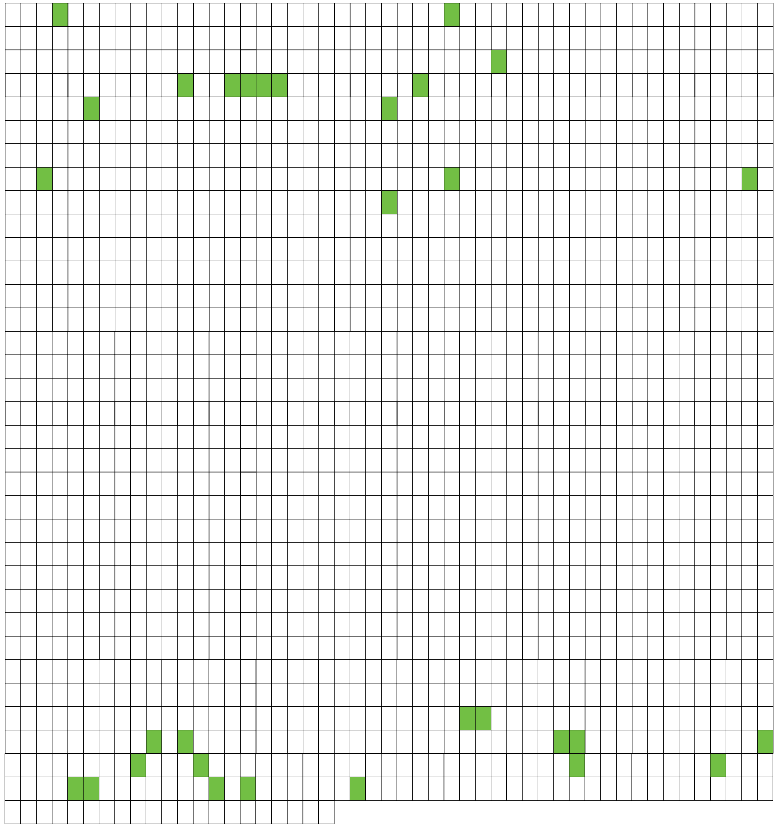
Who dunnit?

Or you have
a bad contract!

		guilty	
		client	implementation
violation	precondition	🦗	
	postcondition		🦗
	invariant		🦗

Contracts in C++20

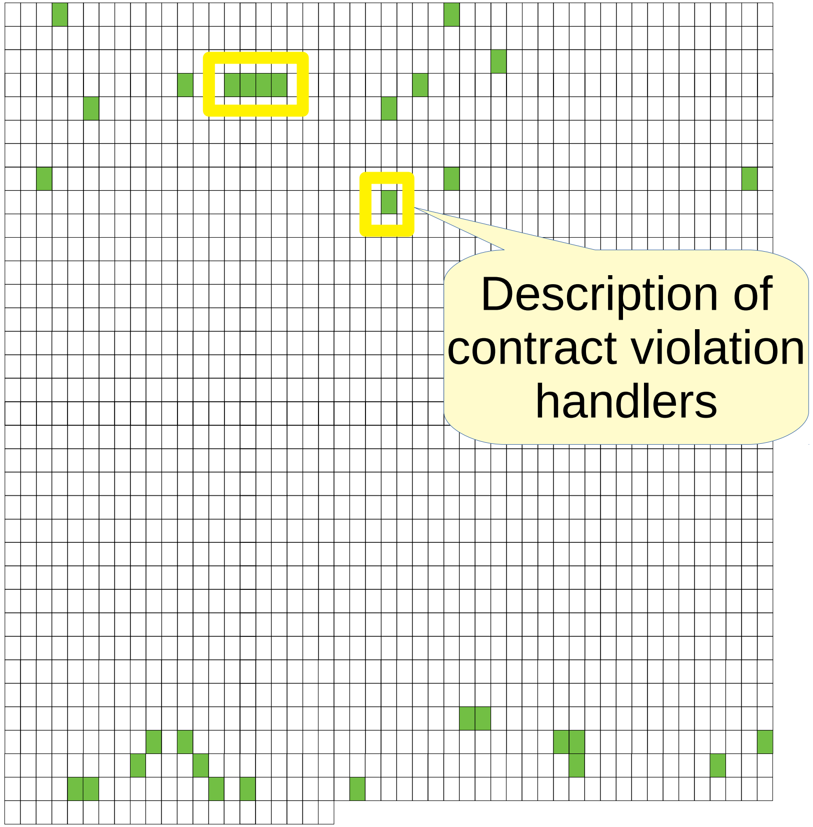
Contracts in C++20



Contracts in C++20

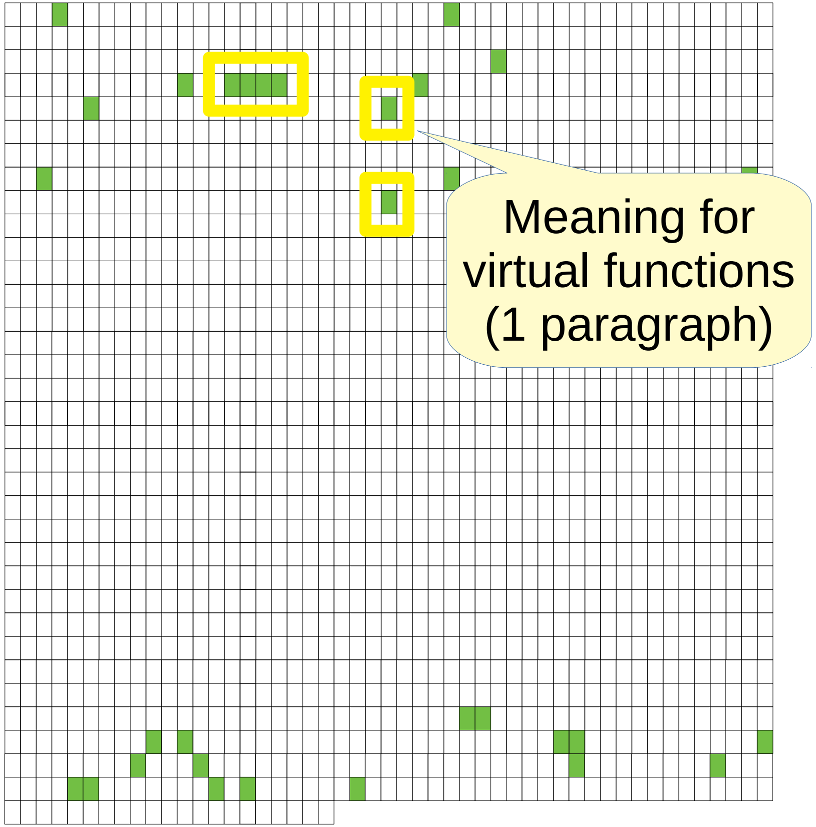
Description of contract attribute declarations and semantics here (4 pages)

Contracts in C++20



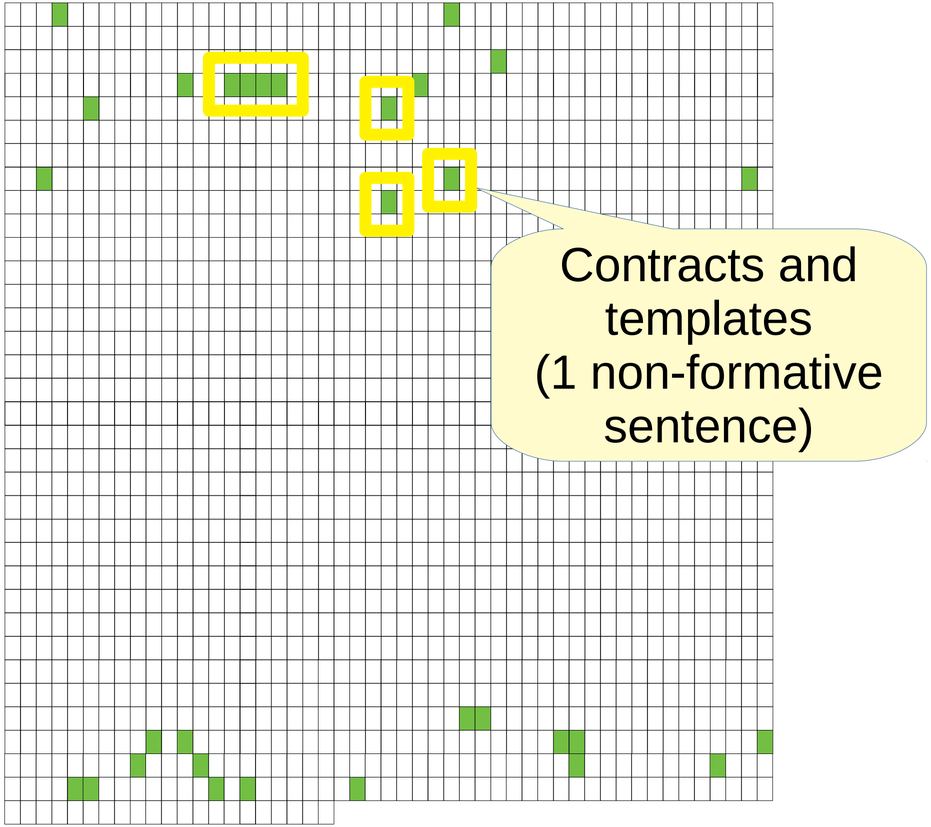
Description of contract violation handlers

Contracts in C++20



Meaning for virtual functions (1 paragraph)

Contracts in C++20



Contracts and templates
(1 non-formative sentence)

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

contract-level:

`default`
`audit`
`axiom`

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[expects *contract-level*$_{opt}$: *conditional-expression*]]

[[ensures *contract-level*$_{opt}$ *identifier*$_{opt}$: *conditional-expression*]]

[[assert *contract-level*$_{opt}$: *conditional-expression*]]

Pre condition

contract-level:

default

audit

axiom

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[expects *contract-level*$_{opt}$: *conditional-expression*]]
[[ensures *contract-level*$_{opt}$ *identifier*$_{opt}$: *conditional-expression*]]
[[assert *contract-level*$_{opt}$: *conditional-expression*]]

Pre condition

contract-level:
default
audit
axiom

```
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects : p != nullptr ]];
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

contract-level:

`default`
`audit`
`axiom`

Optional level

```
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects : p ≠ nullptr ]];
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\texttt{opt}}$: *conditional-expression*]]
[[`ensures` *contract-level*$_{\texttt{opt}}$ *identifier*$_{\texttt{opt}}$: *conditional-expression*]]
[[`assert` *contract-level*$_{\texttt{opt}}$: *conditional-expression*]]

contract-level:

`default`
`audit`
`axiom`

Optional level

```
template <typename T>
void func(std::unique_ptr<T> p)
[[ expects axiom : p != nullptr ]];
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

Post condition

contract-level:

`default`
`audit`
`axiom`

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

`http://eel.is/c++draft/dcl.attr.contract#syn-1`

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

Post condition

contract-level:

`default`

`audit`

`axiom`

```
template <typename T>
T prev(T v)
[[ expects : v > 0 ]]
[[ ensures audit r : r + 1 == v ]];
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

`http://eel.is/c++draft/dcl.attr.contract#syn-1`

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[expects *contract-level*$_{opt}$: *conditional-expression*]]
[[ensures *contract-level*$_{opt}$ *identifier*$_{opt}$: *conditional-expression*]]
[[assert *contract-level*$_{opt}$: *conditional-expression*]]

Name for return value to use in conditional expression

contract-level:
default
audit
axiom

Post condition

```
template <typename T>
T prev(T v)
[[ expects : v > 0 ]]
[[ ensures audit r : r + 1 == v ]];
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

Generic assertion

contract-level:

`default`
`audit`
`axiom`

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:

[[`expects` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

[[`ensures` *contract-level*$_{\text{opt}}$ *identifier*$_{\text{opt}}$: *conditional-expression*]]

[[`assert` *contract-level*$_{\text{opt}}$: *conditional-expression*]]

Generic assertion

contract-level:

`default`
`audit`
`axiom`

```
for (auto p : pointers) {
  [[ assert axiom: p ≠ nullptr ]];
  func(p);
}
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Contract attributes in C++20

9.11.4.1 Syntax [dcl.attr.contract.syn]

1# Contract attributes are u ions, postconditions, and assertions for functions.

There are no class invariants!

contract-attribute-specifier:

```
[ [ expects contract expression ] ]
[ [ ensures contract-level_opt identifier_opt : conditional-expression ] ]
[ [ assert contract-level_opt : conditional-expression ] ]
```

contract-level:

```
    default
    audit
    axiom
```

An ambiguity between a *contract-level* and an *identifier* is resolved in favor of *contract-level*.

http://eel.is/c++draft/dcl.attr.contract#syn-1

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() ⩾ 0 && size() ⩽ N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

No support for
class invariants,
so might as well
leave as comment.

Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer();
  // ensures: size() == 0
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  ringbuffer()
  [[ ensures: size() == 0 ]];
  int size() const;
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() < = N
  ringbuffer()
  [[ ensures: size() == 0 ]];
  int size() const;
  const T& back() const;
  // requires: size() > 0
```

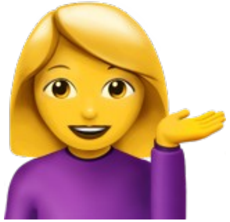
```
<source>:6:15: error: use of undeclared identifier 'size'
  [[ ensures: size() == 0 ]];
```

```
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && si
  ringbuffer()
  [[ ensures: size() == 0 ]];
  int size() const;
  const T& back() const;
  // requires: size() > 0
```

Contract attributes are declarations that can only refer to identifiers seen earlier.



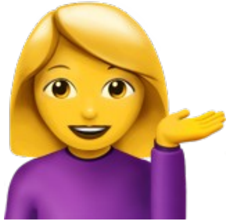
```
<source>:6:15: error: use of undeclared identifier 'size'
  [[ ensures: size() == 0 ]];
```

```
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && si
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Contract attributes are declarations that can only refer to identifiers seen earlier.



Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const;
  // requires: size() > 0
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const;
  // requires: size() > 0
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t);
  // requires: size() < N
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old back();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]];
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old back();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]];
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]];
  // ensures: size() == old size()+1
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

There is no way
to refer to previous state
so this cannot be
expressed!

Using C++20 contract attributes for ringbuffer

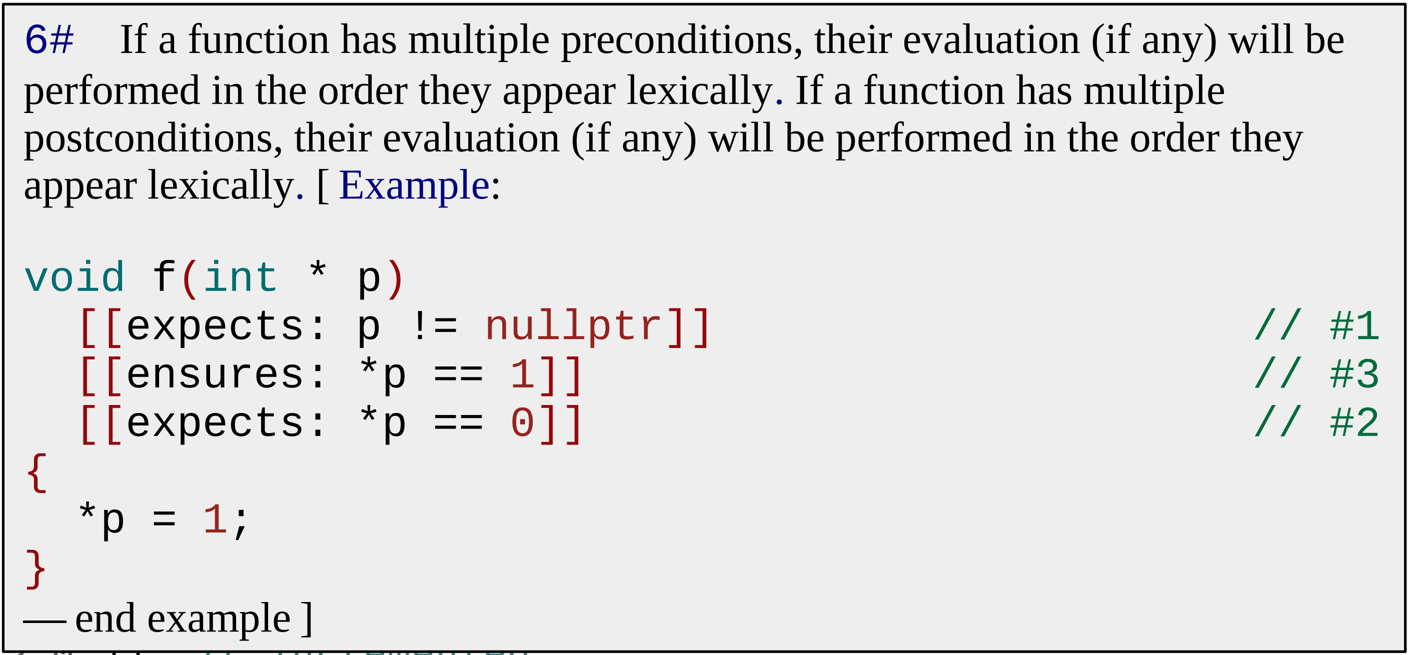
```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]]; // incremented
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]];
  [[ ensures: size() > 0 ]]; // incremented
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T,
class ringbuffer {
public:
  // invariant: size(
  int size() const;
  ringbuffer()
  [[ ensures: size()
  const T& back() con
  [[ expects: size()
  const T& front() co
  [[ expects: size()
  void push_back(T t)
  [[ expects: size()
  [[ ensures: size()
  //          back()
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



6# If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically. [Example:

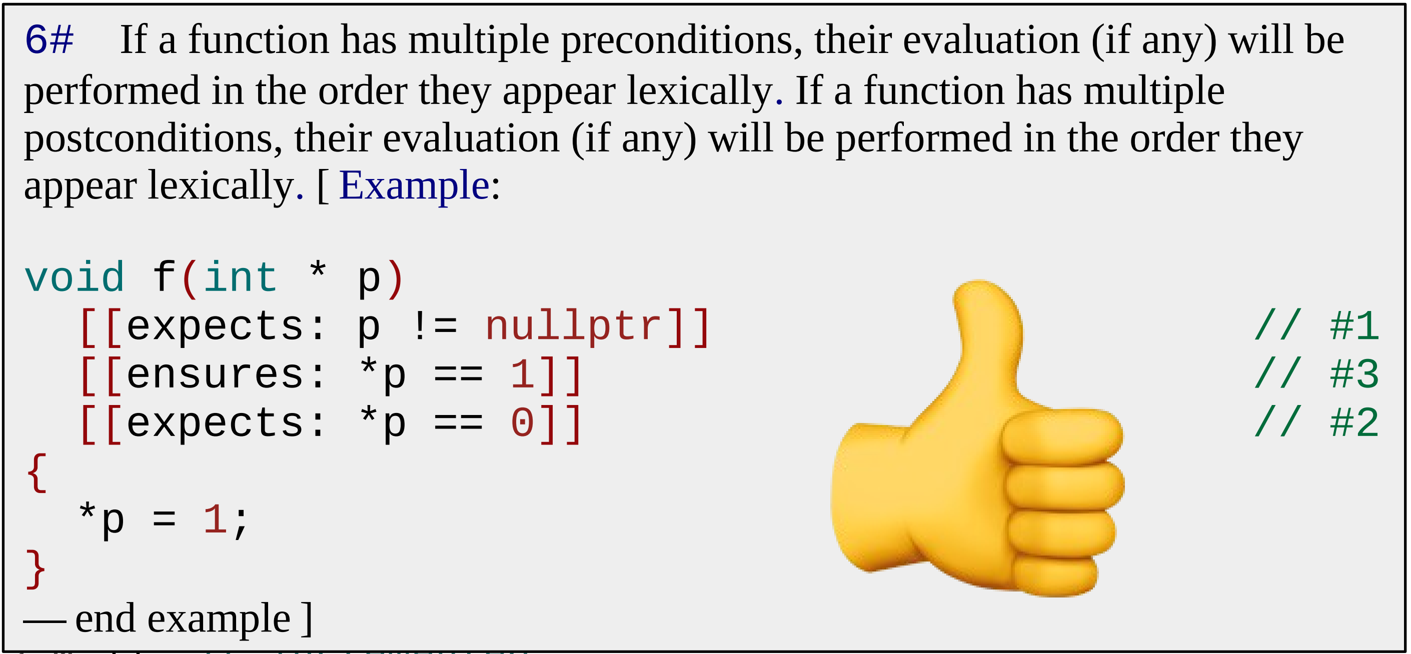
```
void f(int * p)
  [[expects: p != nullptr]]                      // #1
  [[ensures: *p == 1]]                           // #3
  [[expects: *p == 0]]                           // #2
{
  *p = 1;
}
```

— end example]

http://eel.is/c++draft/dcl.attr.contract#cond-6

Using C++20 contract attributes for ringbuffer

```
template <typename T,
class ringbuffer {
public:
  // invariant: size(
  int size() const;
  ringbuffer()
  [[ ensures: size()
  const T& back() con
  [[ expects: size()
  const T& front() co
  [[ expects: size()
  void push_back(T t)
  [[ expects: size()
  [[ ensures: size()
  //          back()
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



6# If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically. [Example:

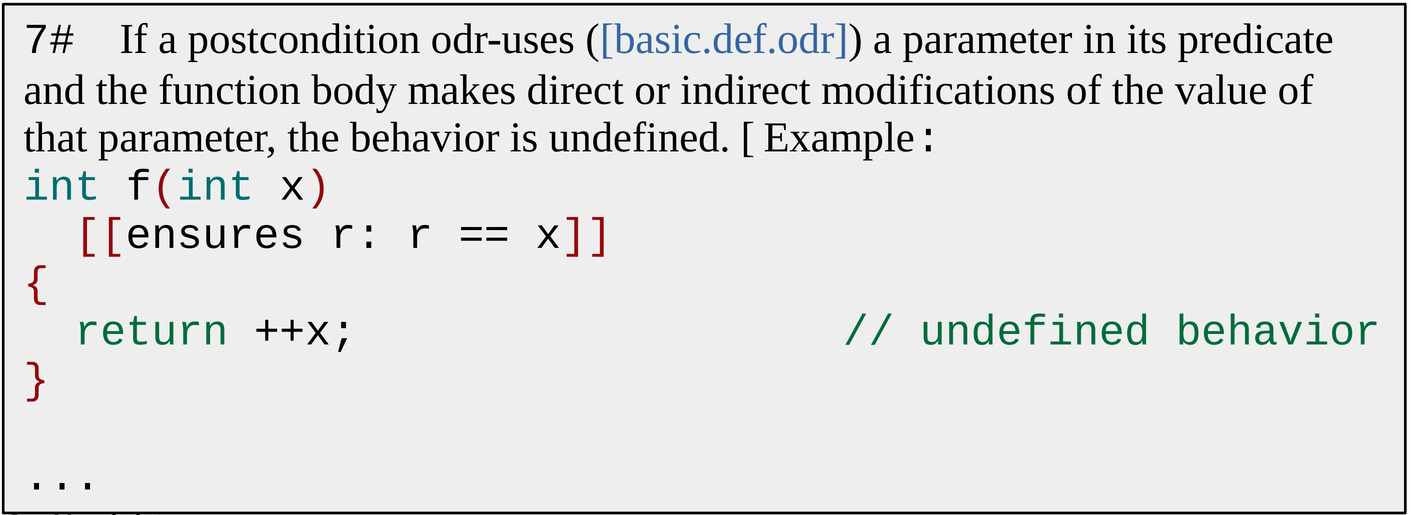
```
void f(int * p)
  [[expects: p != nullptr]]                         // #1
  [[ensures: *p == 1]]                              // #3
  [[expects: *p == 0]]                              // #2
{
  *p = 1;
}
```

— end example]

http://eel.is/c++draft/dcl.attr.contract#cond-6

Using C++20 contract attributes for ringbuffer

```
template <typename T,
class ringbuffer {
public:
  // invariant: size(
  int size() const;
  ringbuffer()
  [[ ensures: size()
  const T& back() con
  [[ expects: size()
  const T& front() co
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]]; // incremented
  //           back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //           return == old front();
};
```



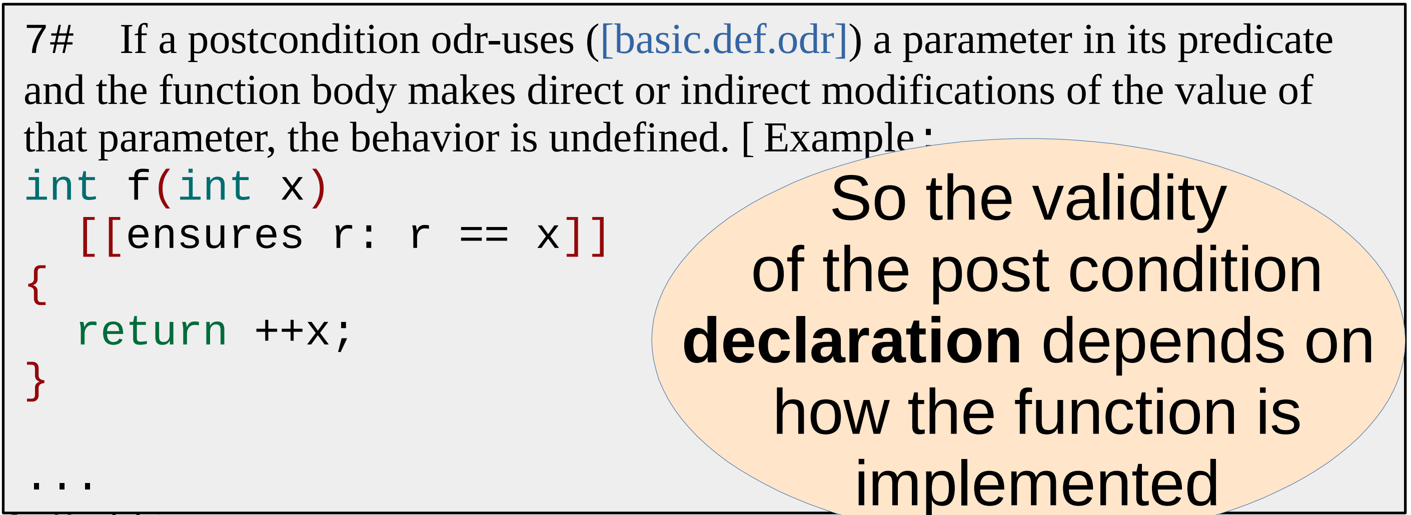
7# If a postcondition odr-uses ([basic.def.odr]) a parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined. [Example :

```
int f(int x)
  [[ensures r: r == x]]
{
  return ++x;                    // undefined behavior
}

...
```

http://eel.is/c++draft/dcl.attr.contract#cond-7

Using C++20 contract attributes for ringbuffer

```
template <typename T,
class ringbuffer {
public:
  // invariant: size(
  int size() const;
  ringbuffer()
  [[ ensures: size()
  const T& back() con
  [[ expects: size()
  const T& front() co
  [[ expects: size()
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]]; // incremented
  //          back() == t
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```



7# If a postcondition odr-uses ([basic.def.odr]) a parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined. [Example :

```
int f(int x)
  [[ensures r: r == x]]
{
  return ++x;
}

...
```

http://eel.is/c++draft/dcl.att ontract#cond-7

So the validity of the post condition **declaration** depends on how the function is implemented

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Potentially dangerous

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front();
  // requires: size() > 0
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front()
  [[ expects: size() > 0 ]];
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```cpp
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front()
  [[ expects: size() > 0 ]];
  // ensures: size() == old size()-1
  //          return == old front();
};
```

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front()
  [[ expects: size() > 0 ]]
  [[ ensures: size() < N ]]; // decremented
  //          return == old front();
};
```

😞

Using C++20 contract attributes for ringbuffer

```
template <typename T, int N>
class ringbuffer {
public:
  // invariant: size() >= 0 && size() <= N
  int size() const;
  ringbuffer()
  [[ ensures: size() == 0 ]];
  const T& back() const
  [[ expects: size() > 0 ]];
  const T& front() const
  [[ expects: size() > 0 ]];
  void push_back(T t)
  [[ expects: size() < N ]]
  [[ ensures: size() > 0 ]] // incremented
  [[ ensures: back() == t ]];
  T pop_front()
  [[ expects: size() > 0 ]]
  [[ ensures: size() < N ]]; // decremented
  //          return == old front();
};
```

Cannot express condition with previous state so might as well leave as comment

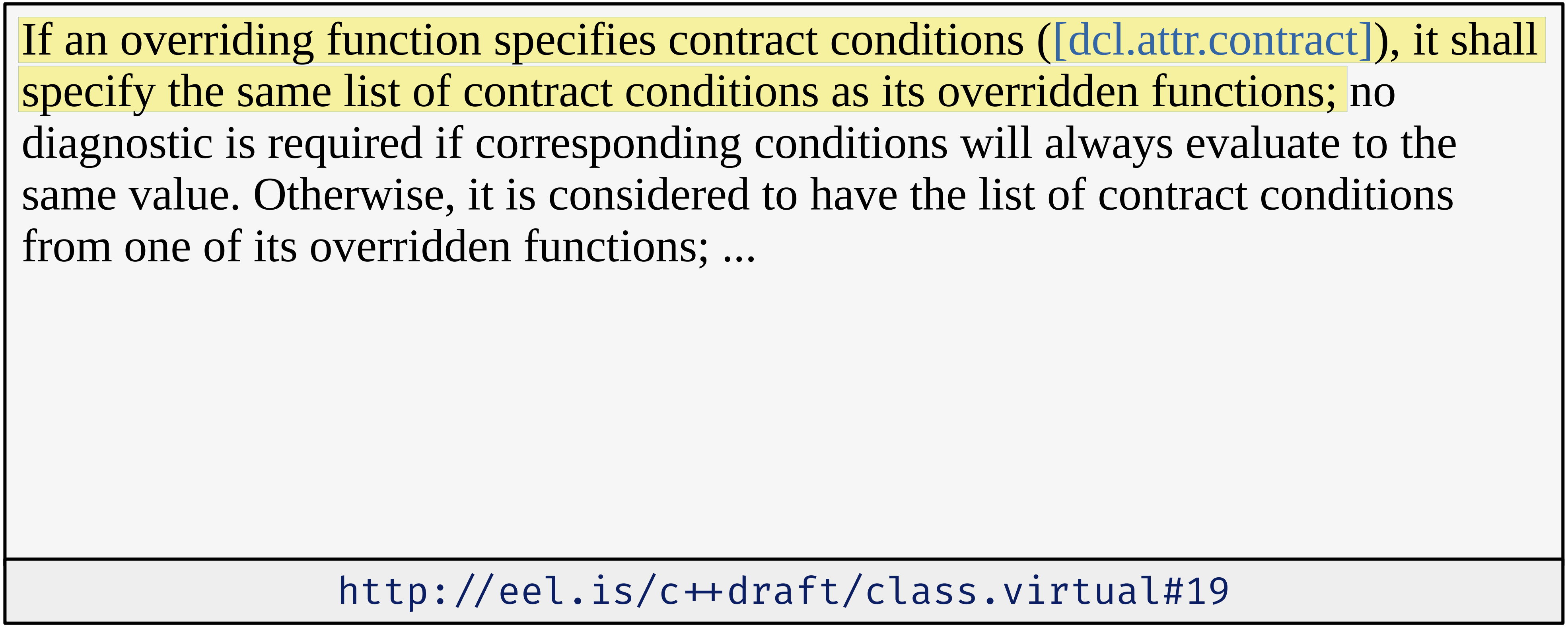
😞

Virtual functions and contracts in C++20

Virtual functions and contracts in C++20

> If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...
>
> `http://eel.is/c++draft/class.virtual#19`

Virtual functions and contracts in C++20



If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...

http://eel.is/c++draft/class.virtual#19

Virtual functions and contracts in C++20

> If an overriding function specifies contract conditions ([dcl.attr.contract]), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; ...
>
> `http://eel.is/c++draft/class.virtual#19`

Function pointers and contracts in C++20

Function pointers and contracts in C++20

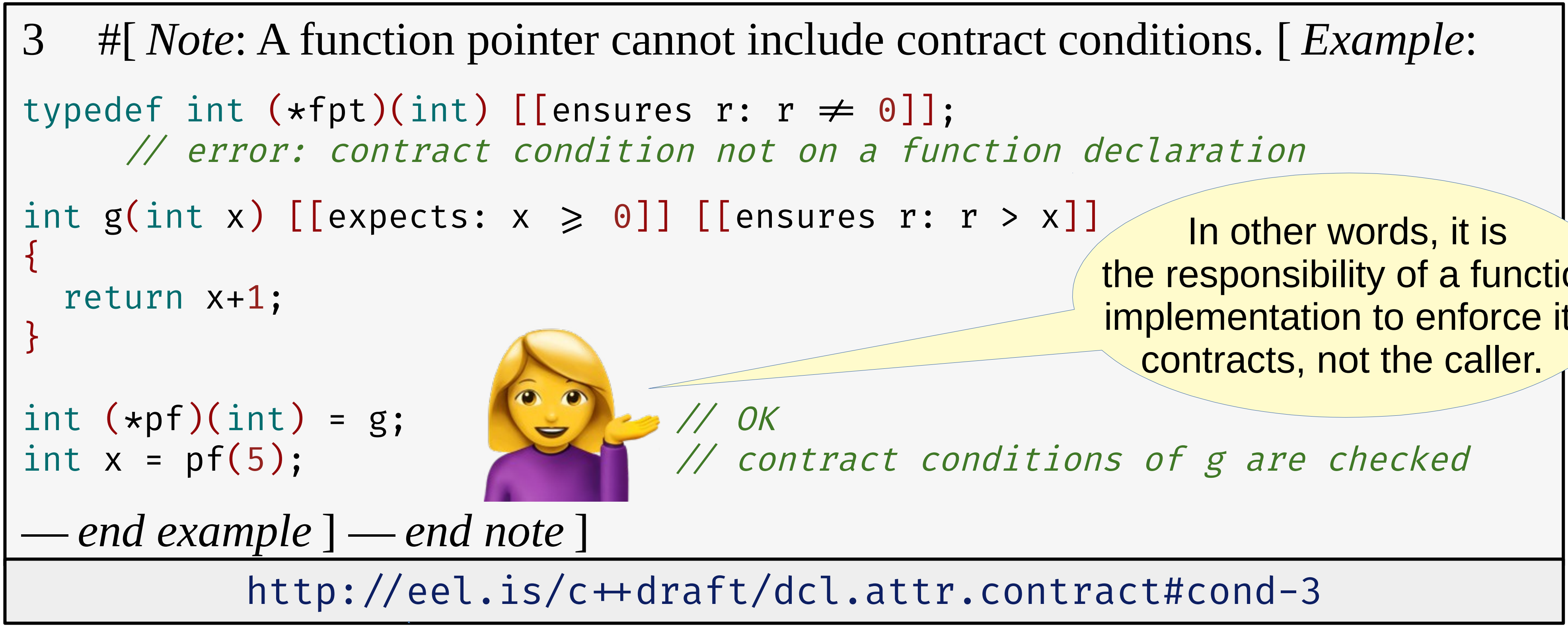
3 #[*Note*: A function pointer cannot include contract conditions. [*Example*:

```
typedef int (*fpt)(int) [[ensures r: r ≠ 0]];
     // error: contract condition not on a function declaration

int g(int x) [[expects: x ⩾ 0]] [[ensures r: r > x]]
{
  return x+1;
}

int (*pf)(int) = g;                   // OK
int x = pf(5);                        // contract conditions of g are checked
```

— *end example*] — *end note*]

http://eel.is/c++draft/dcl.attr.contract#cond-3

Function pointers and contracts in C++20



3 #[*Note*: A function pointer cannot include contract conditions. [*Example*:

```
typedef int (*fpt)(int) [[ensures r: r ≠ 0]];
     // error: contract condition not on a function declaration

int g(int x) [[expects: x ⩾ 0]] [[ensures r: r > x]]
{
  return x+1;
}

int (*pf)(int) = g;                      // OK
int x = pf(5);                           // contract conditions of g are checked
```

— *end example*] — *end note*]

http://eel.is/c++draft/dcl.attr.contract#cond-3

In other words, it is the responsibility of a function implementation to enforce its contracts, not the caller.

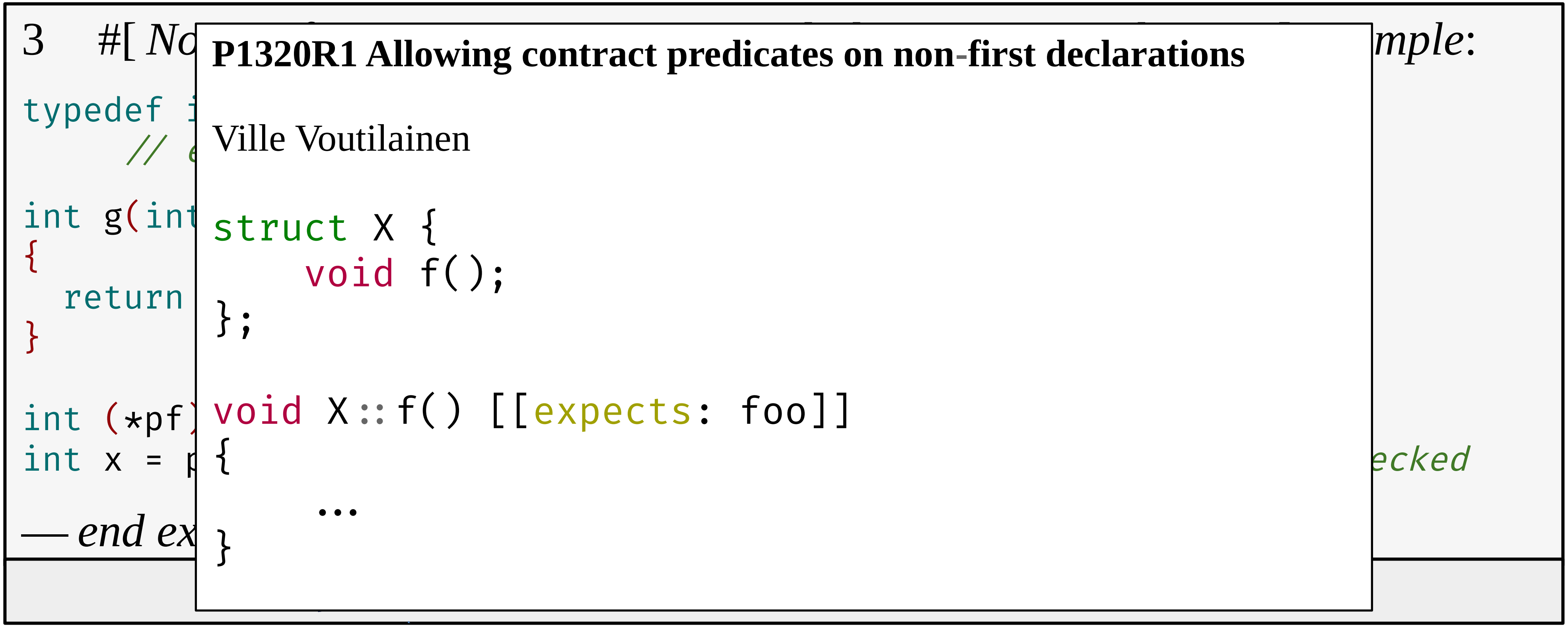
Let’s explore!

`https://github.com/arcosuc3m/clang-contracts`

Fork from clang-6

COMPILER EXPLORER `http://fragata.arcos.inf.uc3m.es/#`

Function pointers and contracts in C++20



3 #[No … mple:

```
typedef i
      // e

int g(int
{
  return
}

int (*pf)
int x = p                                                    ecked
```

— end ex

P1320R1 Allowing contract predicates on non-first declarations

Ville Voutilainen

```
struct X {
    void f();
};

void X::f() [[expects: foo]]
{
    ...
}
```

Policing contracts in C++20

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

`http://eel.is/c++draft/dcl.attr.contract#check-3`

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

3# A translation may be performed with one of the following build levels: off,
default, or audit. A translation with build level set to off performs no checking
for any cking
for defa s
checkin cted,
the buil
impleme ation
units wh ionally-
supported. There should be no programmatic way of setting, modifying, or
querying the build level of a translation unit.

P1421r0 Assigning semantics to different Contract Checking Statements

Andrzej Krzemieński

Allowing different levels for different types of contracts, e.g. audit on preconditions, and off on the others.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

3# A tra ls: off,
default, o checking
for any c ecking
for defau ms
checking lected,
the build
implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

3.7 [defns.cond.supp]
conditionally-supported
program construct that an implementation is not required to support
[*Note*: Each implementation documents all conditionally-supported constructs that it does not support. — *end note*]

http://eel.is/c++draft/intro.defs#defns.cond.supp

http://eel.is/c++draft/dcl.attr.contract#check-3

Policing contracts in C++20

> 3# A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

http://eel.is/c++draft/dcl.attr.contract#check-3

Let’s explore!

`https://github.com/arcosuc3m/clang-contracts`

Fork from clang-6

COMPILER EXPLORER `http://fragata.arcos.inf.uc3m.es/#`

Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

COMPILER EXPLORER http://fragata.arcos.inf.uc3m.es/#

```
-build-level=(off|default|audit)
```

When contracts are violated in C++20

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [*Note*: Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

`http://eel.is/c++draft/dcl.attr.contract#check-5`

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [*Note*: Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

`http://eel.is/c++draft/dcl.attr.contract#check-5`

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to const std::contract_violation) returning void”. The
violation aluates
to false (c ay of
setting or how the
violation
std::co elow. If a
precondit tation-
defined. [rt the
caller site n of the
violation is
violated,
statement

16.8.2 Class contract_violation [support.contract.cviol]

```
namespace std {
  class contract_violation {
  public:
    uint_least32_t line_number() const noexcept;
    string_view file_name() const noexcept;
    string_view function_name() const noexcept;
    string_view comment() const noexcept;
    string_view assertion_level() const noexcept;
  };
}
```

http://eel.is/c++draft/support.contract.cviol

http://eel.is/c++draft/dcl.attr.contract#check-5

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [*Note*: Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

http://eel.is/c++draft/dcl.attr.contract#check-5

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [*Note*: Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

`http://eel.is/c++draft/dcl.attr.contract#check-5`

When contracts are violated in C++20

5# The violation handler of a program is a function of type “noexcept$_{opt}$ function of (lvalue reference to `const std::contract_violation`) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [*Note*: Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

`http://eel.is/c++draft/dcl.attr.contract#check-5`

Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

COMPILER EXPLORER http://fragata.arcos.inf.uc3m.es/#

```
-build-level=(off|default|audit)
```

Let’s explore!

https://github.com/arcosuc3m/clang-contracts

Fork from clang-6

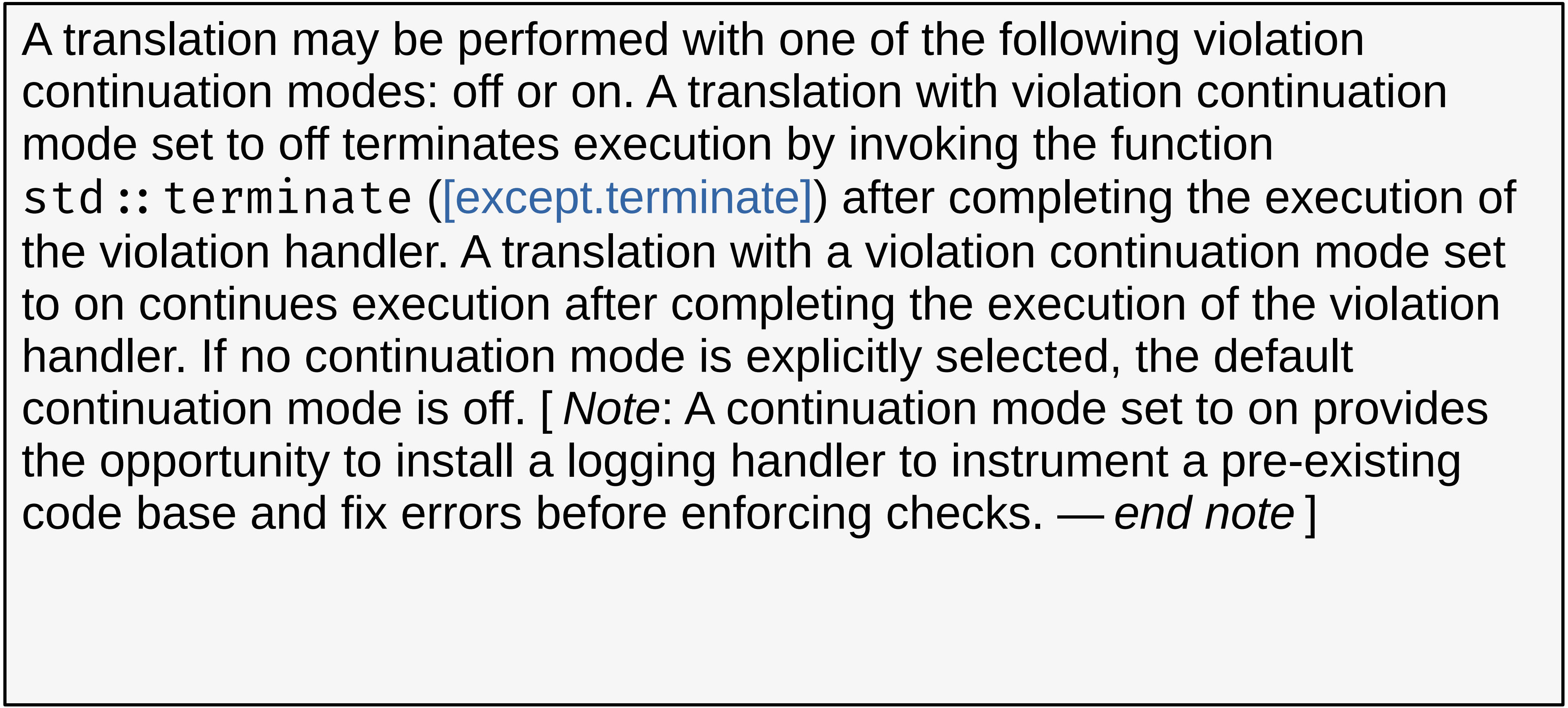
COMPILER EXPLORER http://fragata.arcos.inf.uc3m.es/#

```
-build-level=(off|default|audit)
```

```
-contract-violation-handler=function
```

When contracts are violated in C++20

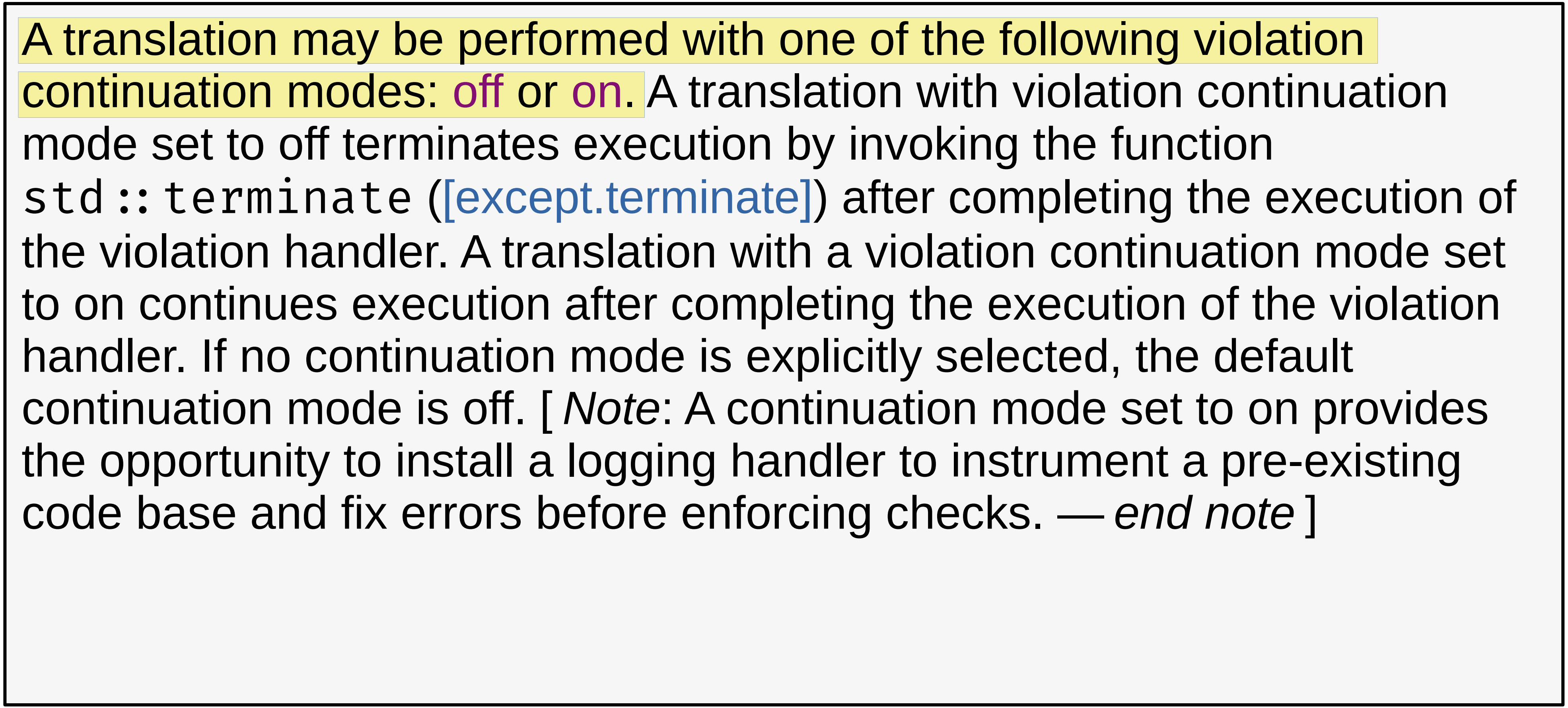
When contracts are violated in C++20



A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [*Note*: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*]

`http://eel.is/c++draft/dcl.attr.contract#check-7`

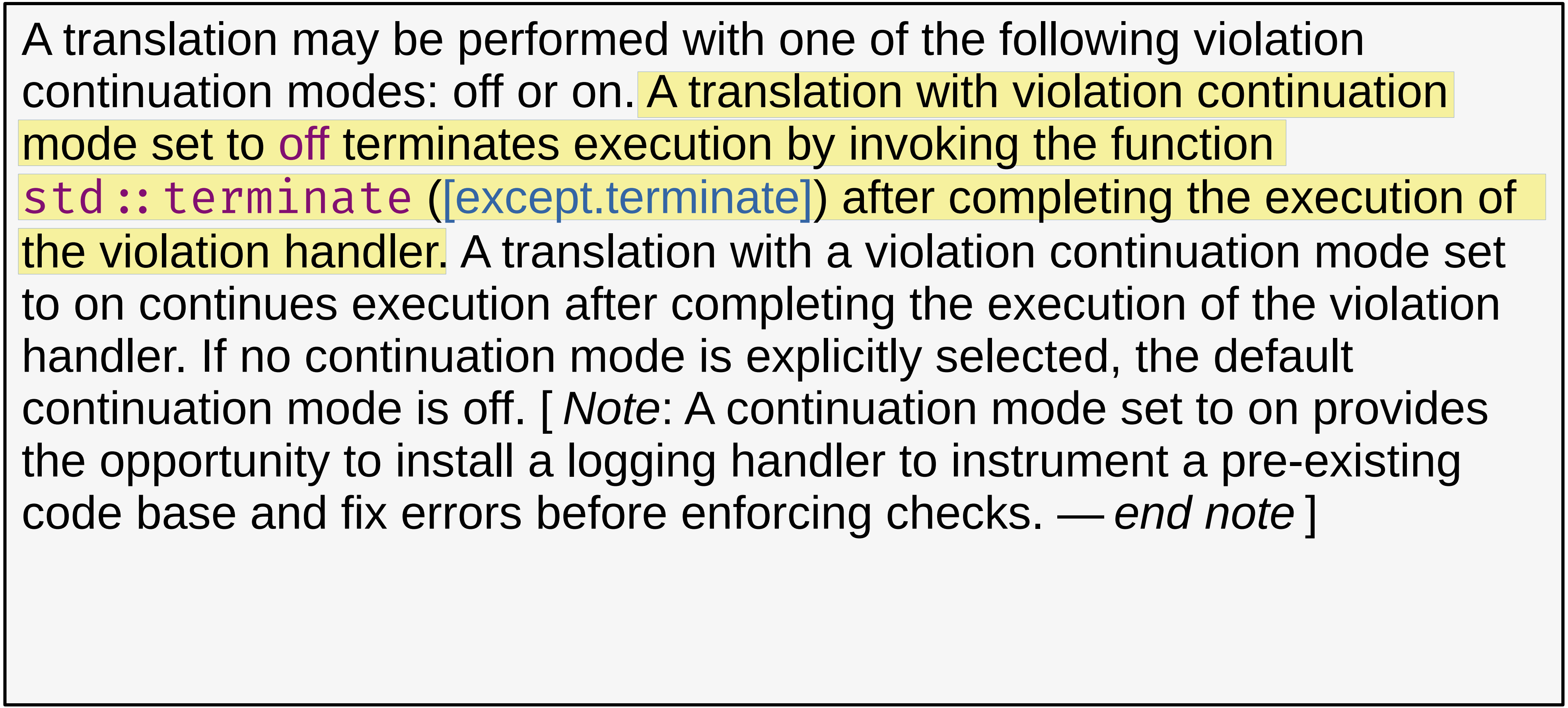
When contracts are violated in C++20



A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [*Note*: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*]

`http://eel.is/c++draft/dcl.attr.contract#check-7`

When contracts are violated in C++20



A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [*Note*: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*]

http://eel.is/c++draft/dcl.attr.contract#check-7

When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [*Note*: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — *end note*]

`http://eel.is/c++draft/dcl.attr.contract#check-7`

When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function

`std::t` … ution of

the viola … de set

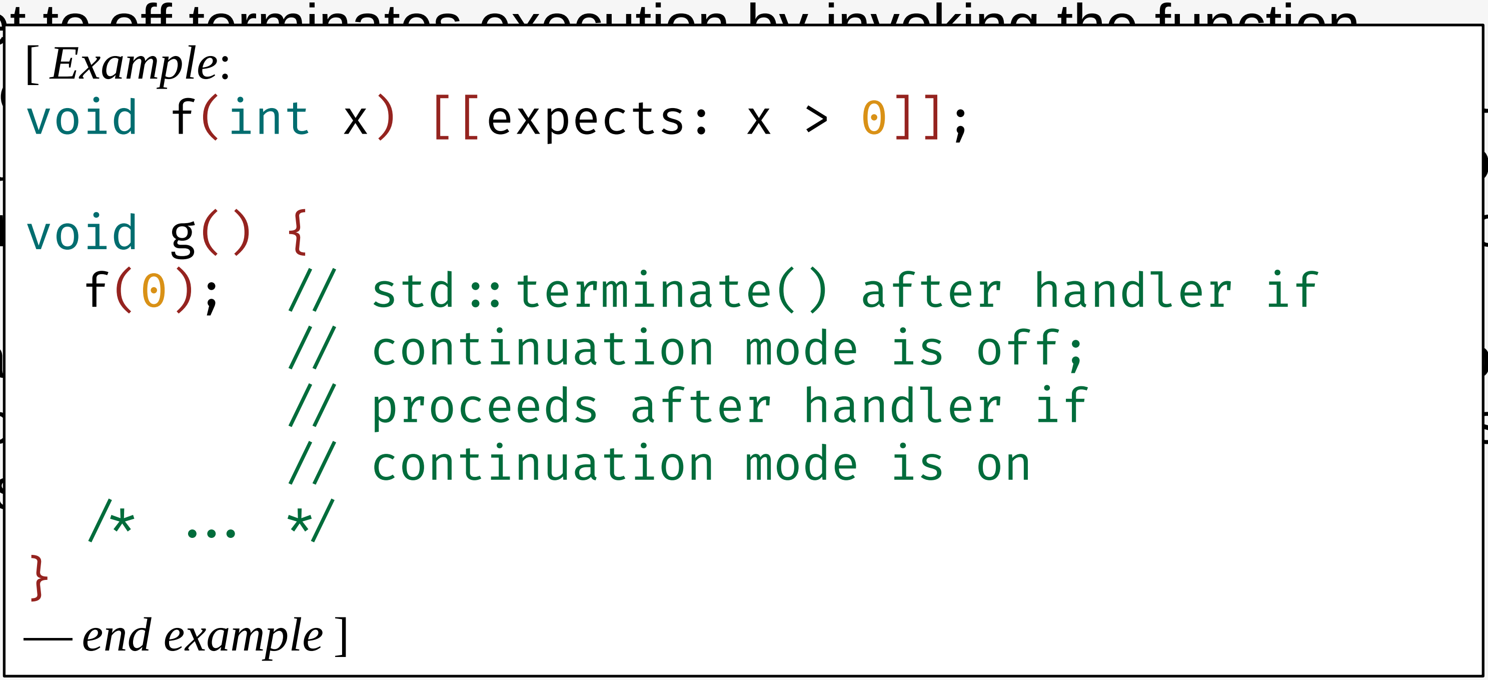
to on co … olation

handler.

continua … vides

the oppo … sting

code bas



[*Example*:

```
void f(int x) [[expects: x > 0]];

void g() {
  f(0);   // std::terminate() after handler if
          // continuation mode is off;
          // proceeds after handler if
          // continuation mode is on
  /* ... */
}
```

— *end example*]

http://eel.is/c++draft/dcl.attr.contract#check-7

When contracts are violated in C++20

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function `std::terminate` ([except.terminate]) after completing the execution of the viola … de set
to on co … olation
handler. …
continua … vides
the oppo … sting
code bas

P1429r0 - Contracts that work

Joshua Bern, John Lakos

Distinguishing between violations that can be safely continued from, and violations that are fatal.

`http://eel.is/c++draft/dcl.attr.contract#check-7`

Programming with Contracts in C++20

🤝

Björn Fahller

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
  - Interesting gotchas:
    - Modifying parameter values, and template specializations comes to mind.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
  - Interesting gotchas:
    - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
  - Interesting gotchas:
    - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20.
  - But it's lackin
  - and post con
  - Interesting go
    - Modifying p ations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.

P1426r0 Pull the Plug on Contracts?

Nathan Meyers.

Argues that the whole idea got wrong and should be scrapped and replaced with something else.

Summary

- Design by contract is a way to clarify the responsibility between a function implementation and its callers.
- Language support is coming in C++20
  - But it’s lacking class invariants
  - and post conditions cannot refer to pre-call state.
  - Interesting gotchas:
    - Modifying parameter values, and template specializations comes to mind.
- Contracts can be used by static analysis tools and the optimizer.
- Configurable levels of contracts, e.g. full in debug builds, only cheap ones in release.
- Prefer to express semantics using the type system, if you can.

Summary

- Desi
  funct
- Lang
  - Bu
  - an
  - Inte
    - M nes to
      m
- Cont
- Conf eap
  ones in release.
- Prefer to express semantics using the type system, if you can.

Play with it!

`https://github.com/arcosuc3m/clang-contracts`

Fork from clang-6

COMPILER EXPLORER `http://fragata.arcos.inf.uc3m.es/#`

Programming with Contracts in C++20

Björn Fahller

bjorn@fahller.se ✉

@bjorn_fahller

@rollbear

#include <C++>